Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingenieros Industriales

Departamento de Automática, Ingeniería Electrónica e Informática Industrial

Master on Industrial Electronics

NOISE-AGNOSTIC SELF-ADAPTIVE EVOLVABLE HARDWARE FOR REAL TIME VIDEO FILTERING APPLICATIONS

Author: **Javier Mora de Sambricio**

Supervisors: **Eduardo de la Torre Arnanz**
Andrés Otero Marnotes

September 2013

Master Thesis

Table of contents

I. INTRODUCTION AND PREVIOUS WORK

1. Introduction

Evolvable hardware is a hot topic in digital electronics. As more complex digital circuits are required for specific applications, the design of said circuits becomes more and more complicated. A solution to this problem is making circuits capable themselves of changing depending on the conditions under which they have to work. In order to do this, an optimization algorithm must find the optimal solution to a particular problem. Evolvable hardware is an approach to this strategy which uses an evolutionary algorithm as the optimization algorithm.

An example of an application of evolvable hardware is a digital signal processor for removing a noise signal from an image. This is a complex task that can have variable requirements, which may not be known when the hardware is designed. Therefore, the system must allow being reconfigured when the working conditions change, which implies the usage of reconfigurable hardware. Furthermore, the complexity of the problem makes it complicated to design the processor functionality, especially if it is intended to do this in an automated manner, which leads to replacing the systematic design of said functionality with an optimization algorithm. Thus, evolvable hardware is a good option for such an application.

One of the problems of evolvable systems is that they need to be trained. This is often done by supplying a training set of data which models the conditions in which the system will operate. However, the generation of this set of data is often done offline, thus reducing the system autonomy.

In this work, a solution for this problem is provided, implemented, and analyzed, together with the obtained results. It will be shown that the system will be capable of evolving without such training reference and, additionally, not being aware of what noise type and levels are present in the image.

This work is a continuation of [1], where an image filtering system based on evolvable hardware was proposed and implemented.

This thesis is structured as follows:

- In this chapter, a theoretical introduction is given and the project targets are established, followed by a description of the state of the art and previous development.
- In chapter II, different possibilities and strategies for obtaining autonomous adaptivity are studied.
- Chapter III describes the development that has been carried out in this project, both in hardware and software.
- Chapter IV shows the results obtained with the developed system and strategies, and assesses these results qualitatively and quantitatively.
- Finally, chapter V closes the thesis assessing what has been achieved and what improvements could be performed and researched.

1.1. Why hardware design

Real time applications, such as video processing, require a high amount of processing power. For example, a 1080p video at 60 frames per second would require a processing speed of near 125 million pixels per second.

An easy way to implement processing tasks is using a software application, which has the advantage of being very flexible and easy to modify. Digital signal processors (DSPs) and computers can easily cope with this task, since they have a high processing power, and have a very high flexibility. However, as processing requirements become more demanding, the processing power provided by these devices is not enough, and the complexity of the system quickly reaches a limit.

In order to have an efficient system, it becomes indispensable to parallelize the computations. An alternative for this is taking advantage of the parallel computation capabilities of the graphics processing unit (GPU) of a computer graphics card as a way to parallelize and accelerate processing tasks. This is known as *general purpose GPU* (GPGPU). However, needing a computer and a graphics card to perform such a task limits the range of applications where these systems could be implemented: size and power consumption can be limiting requirements in embedded applications.

Another approach to this is to use customized hardware rather than general purpose hardware. A way to obtain this is to use *application-specific integrated circuits* (ASICs), which are customized digital circuits that perform a specific task. ASICs allow implementing arbitrary functionality in a very small form factor, but they have

very reduced flexibility: unlike the previously listed alternatives, ASICs need to be designed once and for a static purpose, and their behavior cannot be modified once they have been manufactured. Additionally, manufacturing ASICs has high fixed costs, thus making them unsuited for small production levels.

These issues are solved by *complex programmable logic devices* (CPLDs) and *field-programmable gate arrays* (FPGAs). These devices contain logic gates that can be arbitrarily configured and interconnected in order to implement arbitrary digital circuits. These devices can be reconfigured several times, so that different functionalities can be implemented on them during their service life. FPGAs provide higher configuration capabilities, allowing embedding a complete computing system on the FPGA. This is known as *system on programmable chip* (SoPC).

Although FPGAs are not as fast as some processors, they have the advantage of being able to implement massively parallel hardware that can substantially accelerate computation tasks, and being able to also include a small processor for controlling the overall process makes them suited for implementing a complete autonomous system.

1.2. Evolutionary algorithms

Evolutionary algorithms are optimization algorithms based on trial and error and random incremental search. These algorithms are inspired in the natural process of evolution that allows different species of living beings to appear and adapt to the different conditions of the environment. Natural evolution achieves this with a mechanism known as natural selection: each individual in a species generates a certain number of copies of itself during its lifespan. These are not exact copies; they have slight alterations that make them behave differently, but have similar characteristics. The longer an individual lives, the more copies it can make. Although this is affected by a huge number of random factors, the ability of an individual to live longer and make more copies of itself, which will also be able to live longer and make more copies of themselves, creates an unbalance in this random propagation that will result in more desirable characteristics to be perpetuated while less desirable ones will eventually disappear. This is often referred to as *survival of the fittest*.

An interesting aspect of natural evolution is that it does not require any control or explicit specification of techniques required to survive: individuals in a population will simply adapt to the conditions of their environment, and new individuals with new characteristics will appear naturally, without needing an intentional creation of these characteristics. With the same philosophy, evolutionary algorithms aim to find solutions to a certain problem without the developer to explicitly indicate how these

solutions should be, only supplying a method to compare solutions. The evolutionary algorithm will generate random solutions to the problem, evaluate them, compare them, and generate new solutions by picking the best solutions and applying small random changes on them.

In order to compare different solutions, a method known as *fitness function* is supplied. This is a function that maps an individual to a numeric value known as *fitness* based on its performance: individuals that perform better will have a higher (or lower) fitness than those with worse performance. For example, in the case of the implementation of a discrete filter, the evaluation of the fitness function could consist in filtering a known training pattern and comparing the result sample by sample with a golden reference, returning a numeric value such as the *sum of absolute errors* (SAE) or the *peak signal-to-noise ratio* (PSNR). In this case, the fitter individuals would be those with lower SAE or higher PSNR.

The general form of an evolutionary algorithm is:

- Generate a certain number of random solutions (the initial population)
- Evaluate those solutions in order to calculate their fitness
- Until a certain condition is met (e.g. a number of iterations has been reached, or a good enough result has been achieved), repeat:
  - Choose some of the solutions according to their fitness (and possibly some other factors such as a random selection)
  - Generate new solutions by applying certain operations to the selected ones
  - Evaluate the new solutions
  - Remove some of the solutions according to their fitness (and possibly some other factors such as age or a random selection)
- Once the condition is met, return the best solution that has been obtained during the evolution.

A common type of evolutionary algorithm is the *genetic algorithm*. In a genetic algorithm, each solution (*phenotype*) is unambiguously represented by a *genotype*, which is a representation of the variable characteristics of a solution in the form of a sequence of values known as *genes* (usually bits, but sometimes other data types are used, such as integers or reals). The set of all possible genotypes is known as the *genotype space*.

In a genetic algorithm, the evolution is performed on the genotypes, which are then implemented in order to evaluate the fitness of the corresponding phenotypes. The

process to generate new genotypes involves two operations known as *genetic operators*:

Crossover: Recombination of the genes of two (or more) genotypes, usually by cutting both genotypes at the same random point (the *crossover point*), and generating a genotype formed by the first part of the first genotype and the second part of the second genotype. This is known as one-point crossover.

Mutation: Modification of a genotype by randomly choosing a certain number of genes and changing them, either by flipping them (in the case of bits), adding or subtracting a random value (in the case of reals), or replacing them with a random value. The number of genes that is changed is known as *mutation rate*.

Genetic algorithms usually perform a crossover over two genotypes and then apply a mutation to the result, although the crossover step can be omitted for simplified genetic algorithms.

The abstract representation of the fitness function over the genotype space is known as *fitness landscape*. In order for a genetic algorithm to perform satisfactorily, small changes in the genotype should cause small changes in the fitness, and thus the fitness landscape will be *smooth*, having few local maxima (peaks) and minima (valleys). [Figure 1] This will make it easy for the evolutionary algorithm to reach the absolute maximum through small changes.

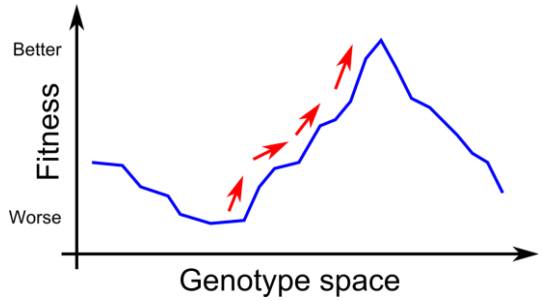


Figure 1. Abstract representation of a smooth fitness landscape. An evolutionary algorithm will easily "climb" to the global maximum in short steps.

If small changes in the genotype may cause big changes in the fitness, the fitness landscape will be *rugged*, having many local maxima and minima, making it difficult for the evolutionary algorithm to reach the absolute maximum: it will likely get stuck in a local maximum, from where it will be impossible or very difficult that the evolution jumps out of there and explores different alternatives. [Figure 2] As a result, the evolution will stall at a suboptimal solution. To avoid this effect, it is a good idea to repeat the evolutionary process several times in case one of the

evolutions got stalled at a suboptimal solution, or modify the evolutionary algorithm so that it detects this condition and performs some action to break out of it.

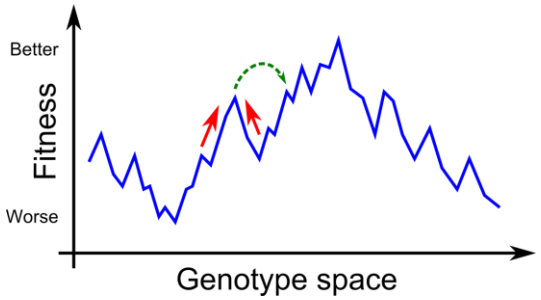


Figure 2. Rugged fitness landscape. It will be hard for an evolutionary algorithm to jump out of the local maximum, and will likely get "stuck" in it.

1.3. Reconfigurable hardware

Reconfigurable hardware is a design approach based on a digital circuit whose functionality can be changed. One of the ways to achieve this is using FPGAs.

FPGAs are often used for digital circuit prototyping and testing, but due to their versatility, they are starting being used for some applications as part of a final product. This way, the product will not only be able to update or change its internal software or firmware, but also improve or modify its hardware capabilities.

Some FPGAs go one step further, and are able to set the configuration of part of the FPGA from the FPGA itself while the rest of it continues working, which allows the development of autonomous reconfigurable systems that do not rely on an external device to change their functionality. This is known as *autonomous dynamic partial reconfiguration of hardware* (DPR). Circuits with such capability are said to be *self-reconfigurable*, which is a desirable condition or prerequisite for evolvable hardware.

The main advantage of being able to reconfigure hardware and not only software is that, whereas a software program can be made very big and complex, it is usually limited by the fact that its execution is sequential, and thus a big program will take a long time to execute. In addition, executing a program on a processor has an overhead due to the time spent decoding the instructions, fetching the variables, and storing the results. Hardware, on the other hand, can be parallelized, which means that several operations can be performed at the same time, with digital circuits tailored for a specific application rather than a general purpose one, potentially

resulting on very shorter execution times, especially when a huge amount of operations has to be performed.

The problem with hardware is that it is often a static design with a fixed functionality. Reconfigurable hardware addresses this problem by letting the system change its functionality by implementing several circuits and allowing it to choose the output of one of these circuits. This can be achieved by implementing all the circuits and selecting one of the outputs through a multiplexor. This is done in the *arithmetic logic units* (ALUs) that can be found on microprocessors, and some implementations of evolvable hardware such as the *virtual reconfigurable circuits* (VRCs) [2] shown in Figure 3.

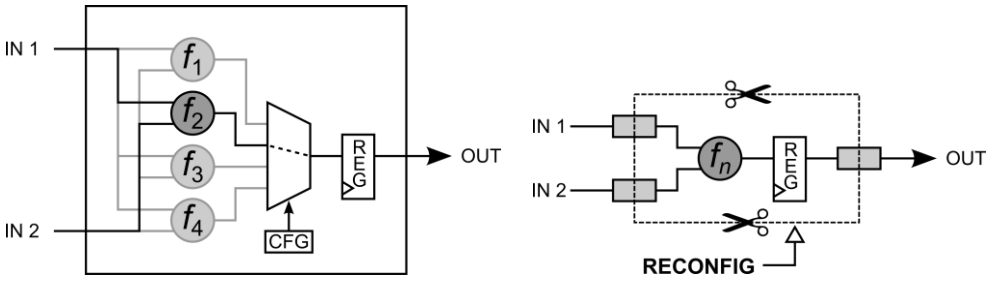


Figure 3. Virtual reconfigurable circuits (left) use a multiplexor to change its functionality. Dynamic partial reconfiguration (right) does this by reconfiguring a region of the FPGA.

Rather than this, using DPR to implement reconfigurable hardware has several advantages. First, implementing a circuit for a single functionality and replace it with another circuit using DPR rather than having all possible circuits synthesized at once implies less resource usage on the FPGA available logic, which may also have a positive repercussion in power consumption. Second, including a multiplexor in the circuit involves increasing the latency, which means that the circuit will have to be more segmented or work at a lower frequency, so it is an advantage to get rid of it.

It has to be noted, however, that DPR has a time overhead when the reconfiguration is performed, but this is not a problem if said reconfiguration is not often performed, and allows the final circuit to be faster once the reconfiguration process has finished.

1.4. Evolvable hardware

Evolvable hardware is an application of reconfigurable hardware in which the reconfiguration is driven by an evolutionary algorithm. This makes evolvable

hardware able to autonomously adapt to different problems without requiring any systematical knowledge on how to implement a solution.

An advantage of choosing evolutionary algorithms for DPR over other optimization algorithms is that these algorithms are incremental, and therefore the phenotype (this is, the processing circuit) will be affected by a small amount of changes. This means that the reconfiguration process will only have to be performed on a few elements of the phenotype, thus achieving shorter reconfiguration times than if all the phenotype had to be reconfigured.

Usage of evolvable hardware is divided in two stages: a training stage, in which the system is optimized for a particular set of requirements; and a mission stage, in which the system has been optimized and can be used to develop the task it was optimized for. Performing a good training stage is crucial for obtaining good results during the mission stage.

The training stage can be performed in two ways: *online*, where the evolutionary algorithm runs in the same device that is going to implement the hardware, and *offline*, where the evolutionary algorithm is executed in an external device such as a computer which will eventually configure the device. An autonomous system does not depend on any external systems to work, and therefore can only use online evolution.

Additionally, the method for evaluating a solution to calculate its fitness can be classified as either *intrinsic* or *extrinsic*. On intrinsic evolution, the evaluation is performed by configuring the hardware with the solution to be tested and evaluating its output, whereas extrinsic evolution uses a software model of the hardware to simulate its behavior. The advantage of intrinsic evolution is that it uses the hardware directly, thus performing the evaluation faster and reflecting any discrepancies the real hardware may have over the theoretical model.

1.5. Aim of the project

This work is a continuation of [1], where an image filter based on evolvable hardware was developed. The aim of the project is to improve that system making it suitable for a real-time application.

In order to achieve this, the system must fulfill these conditions:

- It must be autonomous and self-reconfigurable, not requiring any external control or information to work.
- It must be able to adapt to different conditions of noise type and level, which may not be known a priori.

- It must be fast, being able to evolve in a short time, so that the training stage takes away as little time as possible from the mission stage.
- The hardware must be equipped with an input device such as a camera that will feed the system the images to be filtered.

Until now, the training stage was performed using a noise-free image and adding noise to it, but this cannot be done if the noise type and level for which the filter is being trained is not known. One of the targets of this work is to find a way to perform this stage without having a clean image or a known noise model, using only the images fed to the system. Additionally, the system will be equipped with a camera as a proof of concept of an input device.

2. Previous work

2.1. State of the art

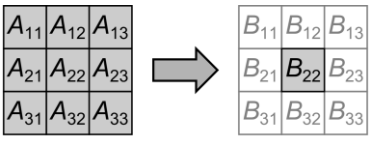
This work covers three main research lines: image processing, evolvable hardware, and dynamic partial reconfiguration of FPGAs.

2.1.1. Image processing

Image processing is a widely researched topic, where both hardware and software methods for processing data are implemented.

One of the methods most frequently used in image and processing is the *window filter*. This filter works by selecting a data window, which in the case of image processing is a rectangular sample from the original image centered on the pixel to be filtered. The new value of said pixel will be calculated as a function of the pixels in the window. This window moves through the input image, calculating the values of all new pixels.

Figure 4 shows an example of a 3×3 window filter with a filter function *f*. Notice that this filter will not be able to reach a 1 pixel border on each side of the image since the window would not be completely inside the image and therefore some of the needed pixels would be unavailable. This is frequently solved by either extending the input image or by making the output image smaller than the input one (in this case, two pixels smaller in each direction).



$$B_{22} = \mathbf{f}(A_{11}, A_{12}, A_{13}, A_{21}, A_{22}, A_{23}, A_{31}, A_{32}, A_{33})$$

Figure 4. 3×3 window filter with a certain function f *filtering the pixel (2,2) of an image*

A filtering function often used on this type of filters for noise removal is the *median*, this is, the output pixel will be found by sorting the 9 (or more) pixels from the window and selecting the middle value, thus discarding outliers. This filter is known as *median filter*, and is specially indicated for filtering noise such as *salt and pepper* noise, in which a few pixels have been largely altered. [3] Although better

filters have been developed, this filter is often used as a reference to compare other filters.

A problem with median filters is that they tend to degrade the image, making the result look blurry, as can be seen in Figure 5. Additionally, this filter is complex to implement, and its complexity grows as larger windows are used. [4]

Figure 5. Image (left) corrupted with salt and pepper noise (middle) and filtered with a 3×3 median filter (right), which tends to blur the image

There are more sophisticated filters, such as the *adaptive median* [5], *switching median* [6], and *weighted order statistics* [7] filters, which perform an analysis of the image in order to determine which pixels are corrupted and require being filtered. These filters can achieve very good results, but are often implemented in software and very complex, leading to very long filtering times [8], making them unsuited for real time processing tasks. A solution is to implement such systems in hardware, as shown in [4] with an adaptive median filter; however, these implementations are often very costly in resources.

Some filters, such as the *adaptive order statistic filter*, are parameterized with a set of coefficients. These values are automatically adjusted in order to perform a specific task or work under certain conditions.

This adjustment can be performed by analyzing the input and determining which configuration may be the best suited, or by training the filter. This training process is an optimization problem in which a test pattern is filtered, and a certain output is expected.

Therefore, the training process requires an input image, which in the case of a noise removal filter would be a test image corrupted with noise, and a target or reference image, which would be the test image before being corrupted. Nevertheless, some works [9] suggest using two corrupted images instead of a corrupted and a clean one.

Other approaches to noise removal for real time video take advantage of the similitude between consecutive frames in order to have additional information to reconstruct the original images. In [10], such a system is implemented, additionally

using a motion detection algorithm to predict and compensate the effect of objects in the picture moving between frames.

2.1.2. *Evolvable hardware*

Evolvable hardware is a hot topic on digital circuit research which presents an evolutionary approach to adaptive hardware design.

Some approaches, such as [11], design a digital circuit from scratch by directly using logic gates. However, this is not a very efficient approach: the evolutionary algorithm has to be executed offline and the FPGA reconfigured for each circuit. In addition, the complexity of designing a complete circuit from low level elements such as logic gates makes it complicated for the evolutionary algorithm to find complex circuits.

A more convenient approach is to provide a basic circuit architecture on which the evolutionary algorithm will operate. This architecture consists on a structure of interconnected *processing elements* (PE), which is implemented beforehand together with a library of processing functions, so that the evolutionary algorithm only has to set a small number of parameters determining which functionality does each processing element execute and how these processing elements are interconnected.

An example of such architecture is the one known as *Cartesian genetic programming* [12] shown in Figure 6, in which processing elements are arranged in a rectangular grid, and each PE reads data from two arbitrary PEs situated to its left (not necessarily the column immediately to the left, but any PE to the left of the current column, or directly an input).

This approach is used in [2], which makes use of multiplexors to determine how the PEs are interconnected and which functionality does each PE have. These PEs are known as *virtual reconfigurable circuits* (VRCs).

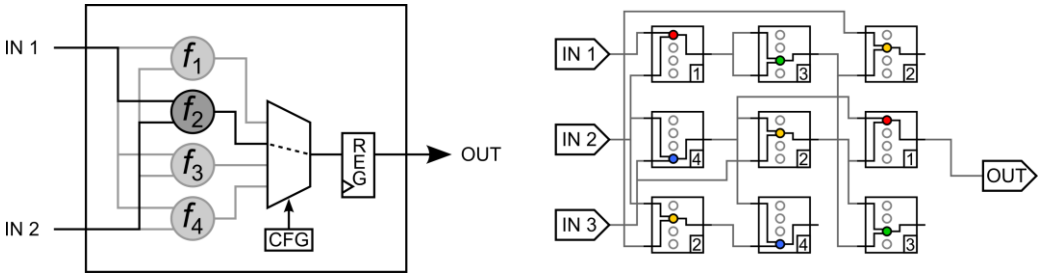


Figure 6. CGP array (right) implemented using VRCs (left), as proposed in [2]

2.1.3. Dynamic partial reconfiguration

Autonomous dynamic partial reconfiguration of hardware is a relatively recent research topic that has been made possible by the inclusion of configuration ports accessible by hardware on FPGAs.

In [13], DPR is used to reconfigure fragments of the FPGA. However, these reconfigurable fragments are not small elements of the FPGA, but very complex circuits that occupy an important part of the FPGA area.

In order to fix the location of inputs and outputs of the partial circuits so that they do not change their position from one partial circuit to another and thus are kept aligned with the static circuit, special circuits known as *bus macros* [14] are used.

Additionally, an alternative reconfiguration method consisting in only reconfiguring some components of the FPGA known as *lookup tables* (LUTs) is mentioned. This approach does not require the reconfiguration of the interconnections, thus avoiding the need of bus macros, but its flexibility is very limited.

In [15], DPR is intended to be used in an evolvable pattern recognition system. However, reconfiguration becomes a bottleneck in speed.

An alternative to the usage of the internal reconfiguration port is presented in [16], where the *shift register LUTs* (SRLs) present in Xilinx FPGAs are used as another way to change the functionality of the FPGA. However, this solution is also limited to the reconfiguration of the LUTs, not allowing reconfiguring the interconnections.

2.1.4. FPGA availability

Among the two main FPGA vendors, Xilinx and Altera, only Xilinx implemented a port and tools for autonomous dynamic partial reconfiguration (although Altera has recently released the Stratix V family featuring fine grain partial reconfiguration), so Xilinx has been the platform of choice for development.

Reconfiguration process of a Xilinx FPGA

Modern Xilinx FPGAs provide an internal port that can be used to access and modify the FPGA configuration. This port is called the *Internal Configuration Access*

Port (ICAP). This port is used to send and read configuration commands and settings, and to read and write configuration data.

The reconfiguration process of a Xilinx FPGA is carried out by sending a sequence of commands and data through the ICAP port. [17] Without going into detail, the steps that take place when performing a reconfiguration, either total or partial, are:

- Configuration setup, in which several configuration options are set and checked by sending commands through the ICAP port. One of these options is the address from where data will start being written.
- Configuration data loading, in which data for the FPGA configuration memory is written to the ICAP.
- Configuration end, in which a checksum of the written data is optionally checked, and the FPGA is set to start working.

All these instructions and data are contained in the *bitstream* files used to configure the FPGA. In particular, the configuration data segment found in these files can be used for extracting configuration data for a partial circuit (a *partial bitstream*).

In order to perform autonomous dynamic partial reconfiguration, Xilinx provides a peripheral, the *Hardware ICAP* (HWICAP) [18], which allows communication with the ICAP port directly through the processor (usually a *MicroBlaze* soft-core processor [19]). However, this is only a low level interface, requiring the developer to implement all the configuration options manually. Moreover, it requires the configuration words to be sent one by one by the processor, which can take a long time for sending large amounts of configuration data.

Structure of the configuration memory of a Xilinx FPGA

The configuration memory of most Xilinx FPGAs [17], depicted in Figure 7, is structured as follows:

- The configuration memory contains everything needed to configure the FPGA: logic block configuration, signal interconnections, block RAM data (which will contain an executable program that will run on an embedded processor), and other special functionality parts are all defined in the configuration memory.
- The configuration memory has 4 *sections*: one for interconnect and block configuration, one for memory content, and two special sections. The section that will be used for dynamic partial reconfiguration of hardware is the first one.
- The FPGA is horizontally divided in a series of *rows*.
- Each row is vertically divided in the same number of *columns*. A column contains different logic elements with a certain functionality: custom logic

blocks (CLB), input/output blocks (IOB), block RAM (BRAM), clock resources... Additionally, all columns have the same number of vertically arranged *interconnection matrices* that create connections with the logic elements in that column and other interconnection matrices in the FPGA.

- Configuration data for each column is divided in data packets called *frames*. A frame is the smallest addressable memory in the configuration memory. The first frames of a column (always the same number of them for a certain FPGA) encode information for the interconnection matrices, and additional frames are used to configure the different logic elements in the column. The number of these additional frames varies depending on the type of logic elements in the column.
- Each frame contains a fixed number of *words*. The ICAP port is able to read or write one word per clock cycle.
- The position of a word in the frame determines which element in the column it configures. For example, in a Virtex-5 FPGA, the first two words of each frame in a CLB column refer to the bottommost interconnection matrix and configurable elements.
- The structure of the configuration memory is mostly independent of its position: a circuit on a column could be copied to another identical column of the FPGA by just copying the content of the configuration memory from one column to the other. Similarly, a circuit can be copied to another position on the column by copying, frame by frame, the words from one position in the frame to another.
- **It is not possible to reconfigure less than a frame.** In order to reconfigure a fragment of a column, the data of each frame must be read, overwritten with the new data at a certain position, and written back to the configuration memory. This makes reconfiguring a fragment of a column slower than reconfiguring all of it.

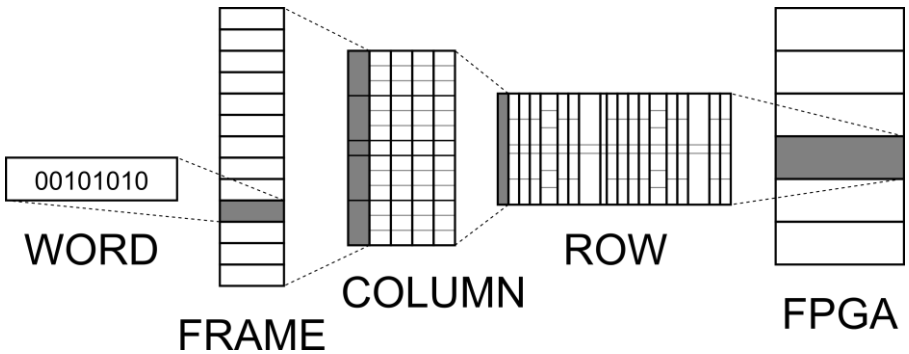


Figure 7. Structure of the configuration memory in the FPGA

2.2. Previous development at CEI

Research on dynamic partial reconfiguration of hardware has been an actively developed topic at Centro de Electrónica Industrial, using Xilinx FPGAs (mostly on the Virtex-5 family, although some work has been done with Spartan-6 and Virtex-II FPGAs).

2.2.1. First steps into DPR

One of the first research works for partial reconfiguration was the development of the *pBITPOS* tool, short for *partial bitstream positioner* [20], which is a software tool for manipulating the bitstream of Virtex II and II-Pro FPGAs that allows placing partial circuits defined in *partial bitstream files* on a complete bistream file, that can later be uploaded to the FPGA with the regular FPGA configuration tools.

Another tool developed at CEI was a custom configuration peripheral that would allow partially reconfiguring the FPGA autonomously. [21] This peripheral is a modification on the HWICAP interface provided by Xilinx [18] with some improvements, such as automatic generation of FPGA addresses. This peripheral is designed to be integrated on an SoPC, being controlled by the processor.

Virtex-5 FPGAs were found to be fitter for development than any other previously used FPGA family because their internal structure is better documented. [17] Particularly, the XUPV5 board is a good choice for the following reasons:

- It features an XC5VLX110T, a Virtex-5 FPGA with high capacity and a very homogeneous internal structure: unlike other Virtex-5 FPGAs, this one does not feature embedded processors and contains large contiguous sections of configurable logic, making it particularly suited for hardware reconfiguration tasks.
- It includes a DDR2 SDRAM memory, which may be used to store the partial bitstreams that will be used to reconfigure parts of the FPGA, and to extend the internal FPGA memory.
- It has a variety of peripherals that can be used for interacting with the FPGA and storing non-volatile data: a CompactFlash card reader, connections for RS232 serial ports, DVI video output, PS/2 keyboard and mouse input, an Ethernet interface...

- It is sponsored by the Xilinx University Program, making it an affordable solution for development at a university. For this reason, it is also a very commonly used FPGA development board in university context, making it easier to share and reutilize work and results.

2.2.2. *The reconfiguration engine: the Enhanced Hardware ICAP*

The partial reconfiguration process is performed by an IP core developed at CEI named the *enhanced Hardware ICAP* (HWICAP) that is controlled by the MicroBlaze processor. This peripheral, used in [22], is an improvement on the previously developed peripheral described in [21], with the following features:

- It handles all data headers and tails needed before and after transmitting the actual reconfiguration data (as seen in 2.1.4)
- It accesses DDR2 RAM memory directly, thus achieving very fast data read speeds
- It abstracts the frame addressing and performs all the needed divisions of the partial bitstream into separate FPGA rows
- It handles writing data to a segment of an FPGA row, providing all needed functionality for ICAP read and write operations

In addition, it has some extra features, such as writing data directly through the MicroBlaze interface, reading a partial bitstream from the configuration memory and storing it in memory, or blanking a region.

Since this core is implemented in hardware, it can achieve very high reconfiguration speeds. Using the HWICAP interface provided by Xilinx, the reconfiguration speed would be around 100 ns per word or more, since data write operations through the PLB are quite slow. However, with the enhanced HWICAP peripheral developed at CEI, reconfiguration speeds of 10 ns per word can be achieved (or even 5 ns if the HWICAP is overclocked at 200 MHz, but this gives problems when reading from the ICAP port).

2.2.3. *Configurable processing architecture: the systolic array*

Evolvable hardware systems are often implemented as a large set of smaller interconnected processing units, whose functionality and connection with other

processing units is decided by the evolutionary algorithm. Having a homogeneous architecture simplifies the reconfiguration process.

A processing architecture suited for evolvable hardware design is the *Cartesian genetic programming* already mentioned in 2.1.2. However, handling interconnections in such a system would be a very complicated task for DPR.

The processing architecture chosen for the filter is known as a *systolic array*. [Figure 8] This is a pipelined parallel architecture consisting of several *processing elements* (PEs) placed in a rectangular grid. Each of these elements computes a value per clock cycle (hence the name *systolic*) by taking the values generated by the top (*north*) and left (*west*) PEs, calculating the result of an operation (usually a simple arithmetic or logic function), registering said result, and sending it to the PEs to the right (*east*) and bottom (*south*). Multiple-input data can be fed directly to the PEs on the north row and the west column, and the result will be extracted from the south-east PE (or, alternatively, any PE on the east column or south row).

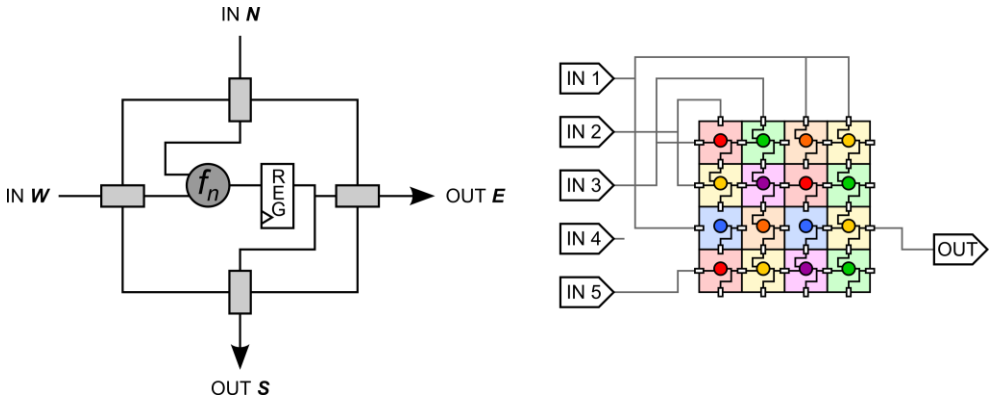


Figure 8. Systolic array architecture, composed of reconfigurable blocks like the one on the left. Compare with Figure 6.

This architecture is less problematic in handling the interconnections, which can be directly implemented as part of the functionality of the processing elements. Therefore, the only additional elements needed for the filter configuration are multiplexors to select which of the multiple inputs go to each north and west PE, and which of the east or south outputs is selected as the filter output; the rest of the configuration is done by using DPR.

2.3. Hardware system

By using the enhanced HWICAP and the systolic array architecture, an implementation of an image filter based on evolvable hardware was elaborated. [1] This platform provided a proof of concept for evolvable hardware systems, allowing the analysis and exploration of features such as self-adaptivity [22] and fault tolerance [23].

The system is designed to train a reconfigurable image filter based on a systolic array. This training is performed by using an evolutionary algorithm that will compare the result of filtering a training image with a reference image. [Figure 9]

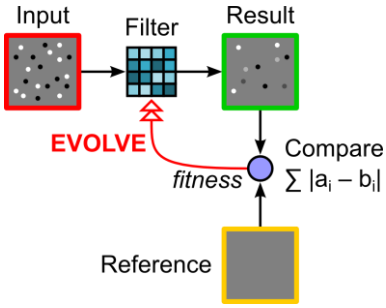


Figure 9. The system evolves a reconfigurable filter using the difference between the output and a reference as fitness

2.3.1. *Structure of the hardware system*

The original hardware implementation was an autonomous system based on an SoPC with the following parts: [Figure 10]

- A processor system
- A configurable image filter based on a systolic array
- A reconfiguration engine

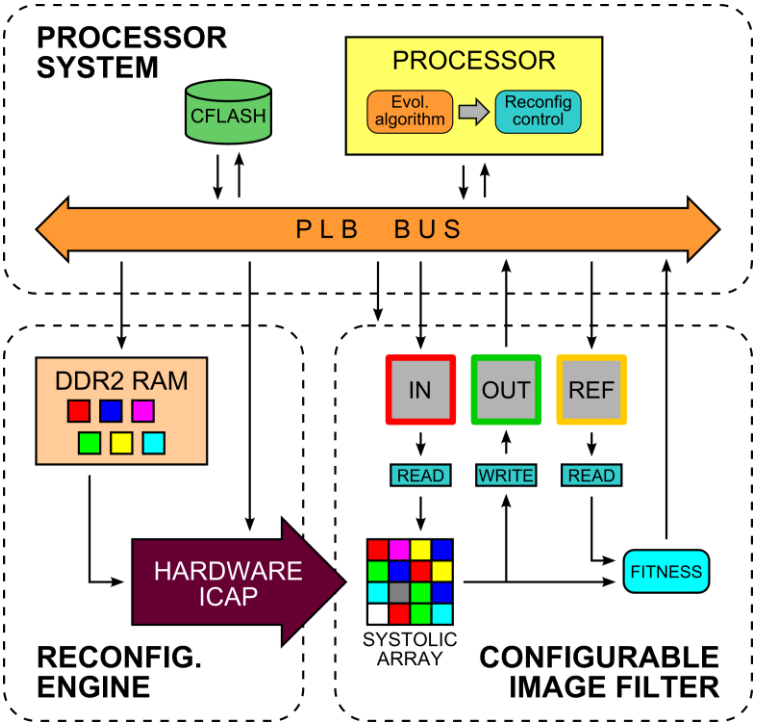


Figure 10. Architecture of the autonomous system based on an SoPC

The processor system

The processor system is the one in charge of executing programs which will implement the evolutionary algorithm and control the other parts of the SoPC. It is composed by:

- A MicroBlaze [19] soft-core processor
- An external storage unit (a CompactFlash memory) for loading and storing the partial bitstream and image files, as well as a register of the evolution progress
- A communication bus (PLB) for interfacing with other components

The reconfiguration engine

The reconfiguration engine is the part which will change the functionality of the filter according to the evolutionary algorithm running on the processor. It consists of:

- An interface with the internal configuration access port (ICAP) of the FPGA, the enhanced HWICAP
- An external RAM memory in which the partial bitstreams of the different functional blocks for the filter are stored, and which can be accessed directly by both the HWICAP and the processor. This memory can also be used by the processor as extra storage space.

The configurable image filter

The configurable image filter is the core of the application. It is based on a systolic array of processing elements placed on a dedicated zone of the FPGA. Additionally, it contains the needed logic for adequately feeding the data to the systolic array, and also features a fitness calculator that compares the filter result with a reference image.

The components of this module are:

- Block RAM (BRAM) memories which store the pixel data of the images and can also be accessed by the processor
- A systolic array (the filter itself)
- A hardware fitness calculator that implements a sum of absolute errors
- Hardware for reading the data from the BRAM, feeding it to the systolic array, and writing it back to the BRAM as well as sending it to the fitness calculator
- A main control block with a state machine that drives the different parts and interfaces with the processor

The images are grayscale raster images of 128×128 pixels, each of them holding an 8-bit value between 0 (black) and 255 (white). These images are loaded from the CompactFlash card, where they are stored as 16384 byte files containing only raw pixel data, with no compression or additional information such as image size or type.

The pixel values are fed to the systolic array by using a 3×3 pixel *moving window*, which is a method frequently used for filtering certain types of noise by comparing each pixel with its neighbors. This window is initially placed on the top left corner of the image, moving one pixel to the right on each clock cycle after it reaches the

right side of the image, after which it moves back to the left side, moving one pixel down.

For the window to be simpler to implement, the process of moving the window from the right back to the left includes 2 clock cycles during which the window is split, and part of it is already on the left while the other part is still on the right. This makes the data captured to have worse quality since these filters rely on the theoretical homogeneity of the pixel values in the window to discard noise in the form of atypical values, so the output data obtained for these instants is discarded from both the result image and the fitness calculation. For a similar reason, the top and bottom rows are discarded. As a result, the filtered image will have a size of 126×126 pixels rather than 128×128.

The systolic array that has been implemented has a size of 4×4 PEs which can implement the following functions (a total of 16 considering the different variations):

- Constant value of 255, regardless of the input
- Identity: output is either north (*N*) or west (*W*)
- Right-shift (*N*/2; *W*/2) and left-shift with saturation above 255 (2N; 2W)
- Maximum (max(*N*,*W*)) and minimum (min(*N*,*W*))
- Addition (*N*+*W*), either modulo 256 or with saturation above 255
- Average ((*N*+*W*)/2)
- Difference (*N*–*W*; *W*–*N*) saturated below 0

Each of the inputs (4 on the north and 4 on the west) takes one of the 9 pixels of the 3×3 pixel window using 9-input multiplexors. The output is selected from one of the 4 east PEs, discarding the 4 south outputs for simplicity.

The array can be configured by reconfiguring the 16 PEs or setting the 8 input multiplexors or the output one. This configuration will be set by the evolutionary algorithm and kept fixed during filtering. Additionally, the delay with which the output should start being captured, which is related with the filter latency, can be configured.

2.3.2. *Description of the filter operation*

The filter operates as follows:

- The processor communicates with the HWICAP in order to make it reconfigure specific PEs in order to obtain the desired filter.

- The processor writes data to some registers in order to configure the multiplexors and set the filter latency (the time the system has to wait before considering the output data valid), and writes to a special register which triggers the filter process.
- The filter starts reading pixel values corresponding to the input image, one pixel per clock cycle, from the BRAM memory, in which the processor will have previously written image data read from a file stored in the Compact Flash card.
- The pixel values are stored in two cascaded 128-stage shift registers. Each shift register will buffer data from the previous pixel row. With the information of the current line and the two previous ones, the 3×3 pixel window can be easily created. 256 pixels (2 rows) must have been read until the shift registers are full and meaningful data can be fed to the systolic array. [Figure 11]
- The 9 pixel values are fed to 8 input multiplexors, each of which will select one of the 9 values according to its configuration and inject it to one of the north or west inputs.
- On each clock cycle, each processing element takes the two input values, performs an operation, and returns the result to the next processing elements. Given the geometry of the filter, the number of clock cycles it takes for an input value to propagate to the output varies depending on how many PEs were crossed.
- The output is collected by the output multiplexor and written to another BRAM memory, excluding a one pixel border from each side of the image, resulting on a 126×126 image.
- Simultaneously, the reference image is read from another BRAM and compared pixel by pixel to the output data. The comparison is done by calculating the absolute value of the difference between two values and adding it to an accumulator, obtaining a similitude measure known as *sum of absolute errors* (SAE).
- Since the filter inputs have different propagation times to the output and their weight on the result is not known, the filter latency that should be expected is not known a priori. For this reason, the SAE is calculated 7 times with a reference image delayed from 0 to 6 clock cycles using a shift register. This is done by implementing 7 separate SAE calculators. Note that only the direct output of the filter is stored as the result.
- Once all pixels have been filtered and stored, and the 7 values of the SAE have been calculated, the filter stops, storing information about the SAE obtained with no extra delay, the best of the 7 SAEs, and the delay for which said SAE was obtained, in three registers that the processor will be able to read.

- The processor will use the information stored in these registers to evaluate the quality of the filter. Additionally, the result of filtering the image can be read from the corresponding BRAM memory and stored on the CompactFlash card.

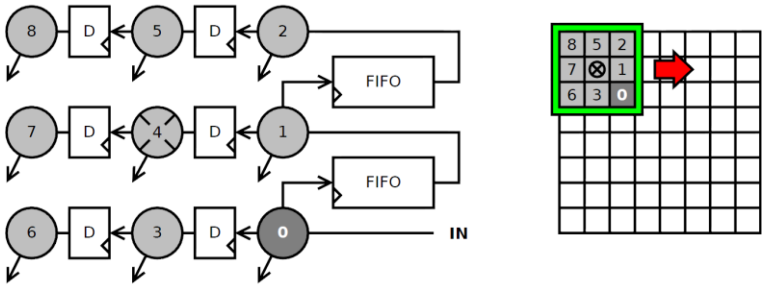


Figure 11. Generation of the 3×3 window using two 128-stage shift registers (FIFOs)

2.3.3. *The evolutionary algorithm*

The evolutionary algorithm chosen was a simplified genetic algorithm with the following characteristics:

- The genotype is a set of 25 integer numbers (*genes*), each of which represents the configuration of a specific multiplexor or processing element in the filter.
- Each gene can have a value from 0 to 15 in the case of processing elements, from 0 to 8 in the case of input multiplexors, or from 0 to 3 for the output multiplexor. Genes only encode the configuration information; which element a specific gene affects depends solely on its position in the genotype.
- The evolutionary algorithm uses a single genotype (the *parent*) to generate 8 new genotypes (the *children*), which are sequentially evaluated. The best of the 9 genotypes (including the parent) is selected as the parent for the new generation. This is known as (1+8)-ES (*evolution strategy*).
- In order to promote the search of new solutions, children have preference over the parent in case of a tie, so if no children achieve a better fitness than the parent but one of them has the same fitness, it will be chosen rather than the parent.
- The only genetic operator used is *mutation*, in which K genes are picked randomly from the genotype and replaced by K new randomly generated values (which will be within the acceptable range for each specific gene). K is known as the *mutation rate*, and is a configurable parameter that remains constant during the evolution.

- There is no guarantee that the selected genes do not overlap nor that the new genes are different from the old ones, thus allowing children to differ from the parent in less than K genes.
- The evaluation process consists in reconfiguring the PEs and multiplexors of the filter with the parameters in a specific genotype, filtering a training image, and calculating the fitness as the SAE with a training reference.
- The evolution starts from a population of 9 randomly generated genotypes, and finishes after a certain number of generations (usually set to 50 000 or 100 000).

This algorithm has proven to work and allow finding very good filters, which outperform the median filter, but often gets stalled after a period of about 20 000 generations, after which improvements in the fitness rarely occur. Further research has shown that it is better to perform several shorter evolutions (for example 5 evolutions of 20 000 generations) and picking the best result than performing a single longer evolution.

The measure employed as fitness is the SAE value calculated by the filter, with low values indicating good filter qualities. This measure is not as good for comparing images as the *sum of squared errors* (SSE), the *peak signal-to-noise ratio* (PSNR), or the *structural similarity* (SSIM), but is simple enough to be calculated in hardware without requiring too many resources, and has proven to perform similarly to the PSNR as a fitness measure for the evolutionary algorithm, since both set as target making the images as similar as possible.

II. ADAPTIVITY OF THE FILTER

3. *Generalizability properties of the filter*

In [1], the system described in section 2.3 was tested with a type of noise known as *salt and pepper noise*, which is the result of replacing random pixels of the image with black or white ones, using always the same test image. The system was tested for noise levels ranging from 5% to 40%. However, no other types of noise were tested.

Nevertheless, the system was not designed for a specific type of noise; so theoretically, the filter could be evolved for any type of noise by just supplying it an image with that noise type and with no noise as training input and reference.

In this section, it is shown that the system can adapt to different problems and is therefore suited for adaptive applications.

3.1. Independence of the image

As can be seen in Figure 12, once a filter has been trained using a specific image, said filter works with other similar images with the same type of noise, not only with the one used in the training stage.

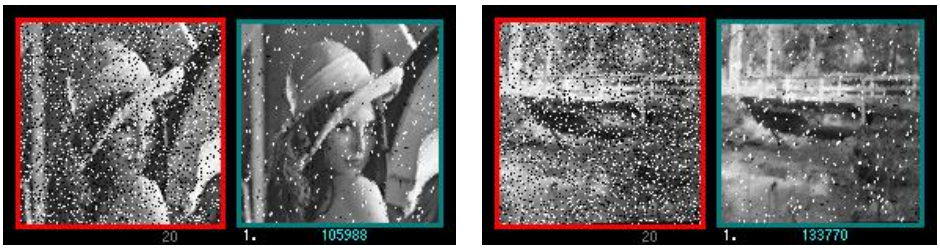


Figure 12. The filter trained with the image on the left has a similar performance with the image on the right.

3.2. Adaptivity to different types of noise

In order to assess the suitability of this system to adaptive applications, the following noise types have been tested, and some results can be seen in Figure 13:

- *Salt and pepper noise*: random pixels of the image are replaced with black or white pixels (values 0 and 255 respectively). Whether each replaced pixel is black or white is randomly selected with a 50% probability. This noise type is relatively easy to filter since it is easy to figure out whether a pixel was corrupted or not: if its value is 0 or 255, it was likely corrupted, since these values are unusual in non-corrupted images.
- *Impulse noise*: random pixels of the image are replaced with random values between 0 and 255 (different shades of gray from black to white). These values are selected with a uniform distribution, so that each of the 256 possible values has a probability of 1/256. This type of noise is more complicated to remove than salt and pepper noise, since in this case it is harder to tell whether a pixel has been corrupted or not.
- *Impulse burst noise*: this noise simulates transitory interruptions of data transmission, and visually looks like white horizontal lines scattered over the image. These lines start at random points and have random length (duration).
- *White additive noise*: unlike the previous types of noise that only affect some of the pixels of the image, additive noise affects all pixels, adding or subtracting a random value to each of them. Being white noise means that this added value is not correlated between pixels and is independent of the old value of the pixel. The distribution this added value has can be either *uniform* or *Gaussian*. In particular, white additive Gaussian noise is also known as *amplifier noise* and is a very common type of noise. The system is not actually expected to deal with a noise of this type, since removing it is a complex task that is out of the scope of this project.

As can be seen, the system gives pretty good results for the first three types of noise, generating filters able to remove most of the noise in only 15 000 generations.

However, the results for additive noise are not as good; the result is a filter that basically blurs the image in order to average the value of surrounding pixels. Therefore, this particular system is not suited for this type of noise, although it is possible that this task can be carried out by enhancing the system with a larger systolic array, a larger window, or more complex processing elements.

Figure 13. Result of evolving with different types of noise. From top to bottom: salt and pepper, impulse, impulse burst, and additive. The filter is trained using the image on the left as input and the one on the right as reference; the result being the one on the middle right.

3.3. Other problems: edge detection

Although the proof of concept of the system is a noise removal filter, it could theoretically be trained to solve arbitrary problems involving a window filter. As an example, the filter has been trained by supplying a clean image as input and, as output, the image passed through an edge detection filter implemented offline in software.

As can be seen in Figure 14, the system is able to generate a filter that imitates the behavior of the software filter.

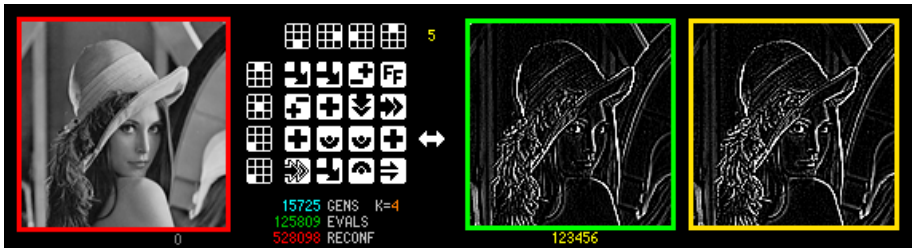


Figure 14. Result of training with a clean input and a reference passed through an edge detection filter. The result is a similar edge detection filter implemented in hardware.

This is an interesting result: the system can also be used as a hardware accelerator that imitates the behavior of arbitrary filters implemented in software.

4. Training without a noise-free reference

4.1. Motivation

The system has proven to effectively adapt to several different requirements for noise filter generation, only needing a training image to which noise has been artificially added, and a reference image which is the same as the training image before adding the noise. However, this is not optimal for autonomous systems implementing a real-time image filter due to the following problems:

- It involves generating a training image. This image could be generated offline and then transmitted to the system, or calculated within the system itself.
- The training image only works for a specific type of noise. Making the system able to adapt to different types and levels of noise would require having either a large library of training images, one for each type of noise; or a noise generator able to generate a different type of noise each time. This adds complexity to the system.
- In order to select the right training image from the library or the right noise generator, it would be necessary to identify the type of noise to be filtered. This is a complex task that would complicate the system too much.
- Additionally, the reference image may differ from the actual images that need to be filtered. A filter optimized for filtering a specific type of noise on a light and sharp image may perform differently on a dark and blurry image with the same type of noise (although it would probably still work, as seen in 3.1). Therefore, it would be preferable to have training images as similar as possible to the ones to be filtered.

Some possible solutions to these problems are:

- Implement a general-purpose noise filter. This filter is not required to work in real time, and could be implemented in software. This filter would be used on an input image, and the result would be used as the training reference in order to evolve the filter to obtain a hardware implementation with a similar behavior to the general-purpose system, but with real time capabilities.
- Develop a system to identify the type and level of noise, and select a pre-defined image or noise generator based on it.
- Make the system use a specific test pattern for which the reference image is known. For the case of a device retrieving images from a camera, this could be done by pointing the camera at a specific location where the test pattern is.

The first two solutions can be implemented online or offline. An online implementation would involve making the system too complex, which may not be desirable for the case of small applications. An offline implementation implies equipping the system with a bidirectional communication module and an external dedicated system for the extra processing tasks. The third solution may not be possible in many circumstances, for example because it is not possible to generate a pattern for the camera or to point the camera at said pattern.

For these reasons, finding an alternative that does not have these problems would be very beneficial for such an autonomous system.

4.2. Chosen solution

As stated before, in order to train a noise removal image filter with an evolutionary algorithm, a training reference image is needed in addition to the training input image. However, said reference does not necessarily have to be perfect. The only thing the evolutionary algorithm needs is an image that will produce better fitness values for filters with better performance, so that each pixel at the output image is as similar as possible to the corresponding pixel at the reference.

Several types of noise, such as *salt and pepper noise* or *impulse burst noise*, affect only some of the pixels of the image, leaving the rest of them unaltered. This means that, although the reference image is corrupted, part of it is still valid and could be used as a reference.

If a noise type of this class is applied to two copies of the same identical image, two noisy images will be obtained as a result, but pixels that are corrupted on the first one may not be corrupted on the second one. [Figure 15] For this reason, a filter could be trained to make the first image look as much as possible like the second image. Unlike the usual operation mode, the filter will not be able to make the input image look like the output image since it will not be able to determine which pixels should be corrupted. However, the evolutionary algorithm will do its best with the pixels that it can actually filter, which are the ones for which the reference is clean. Since the filter applies the same operation to all the pixels independently of their position in the image, pixels that are corrupted in the reference image will be treated the same way as clean ones.

As a result, training the filter with noisy versions of the same image as both the input and the reference will still be a valid way to obtain a filter that removes this type of noise.

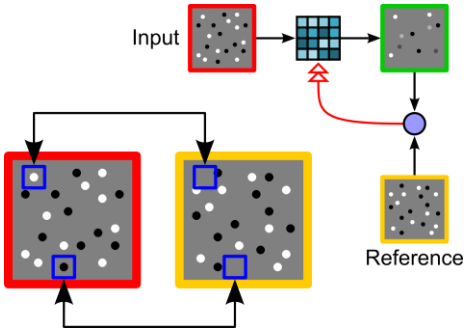


Figure 15. Evolution performed with a noisy image as a reference also works because pixels that are corrupted in the input image are likely not corrupted in the reference image.

4.3. Experimental results using noisy reference images

In order to test this solution, a batch of tests that perform evolutions using frames from a 300 frame video sequence has been launched. These tests are detailed in section 8. Figure 16 compares the results of this solution with those performed with a noise-free reference.

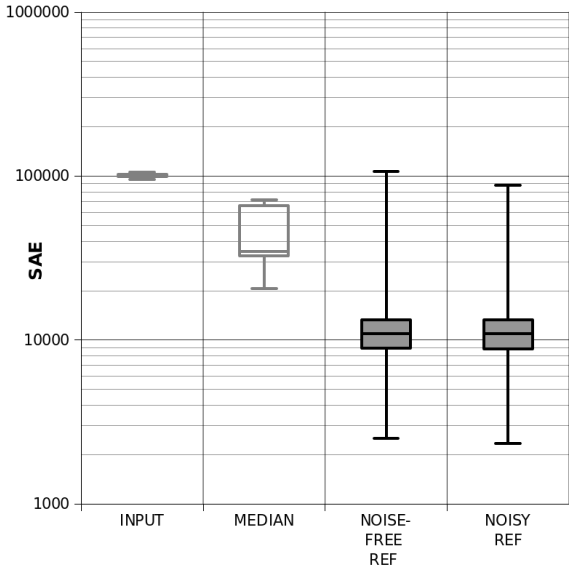


Figure 16. Box plot comparing the results obtained with a median filter, the evolution with a noise-free reference, and the evolution with a noisy reference. The SAE of the input images is also shown.

As can be seen, not only the filter is able to evolve with these two images, it achieves the same results it did with a clean reference.

4.4. Not the case for real-time video

Although this is a very good solution that proves that a clean reference is not needed for training the filter, it has a problem: in general, we will not have two versions of the same image with noise applied to them.

An example application where this method could be used would be filtering noise added by a noisy transmission channel through which the image has been transmitted, and where the image can be transmitted multiple times, but this is not the general case for the real-time application for which this system is intended.

5. *Application to a fully adaptive system for real-time video*

In the previous section, a solution for training filters without having noise-free and noisy images a priori is presented. However, this solution cannot be directly applied for real-time systems such as filtering a sequence of video frames obtained from a camera, since in this case every frame will be unique and two versions of the same image will not be available. In this section, different approaches are considered in order to adapt said solution to video sequences.

5.1. First approach: two consecutive frames

Although a camera takes frames repeatedly and each frame is different from the previous one, it can be assumed that the difference between two consecutive frames will be small. For this reason, it might be possible to use two consecutive frames as a replacement for two identical images.

This solution appeared to be a good idea and was tested on an interactive implementation of the system (detailed in section 6.2), obtaining visually good results. [Figure 17]

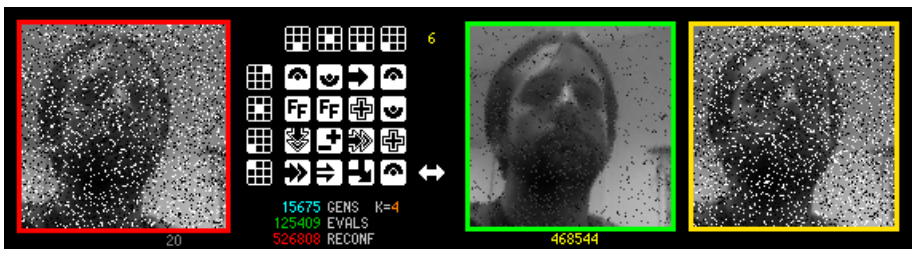


Figure 17. Result (green) of an evolution performed using two consecutive noisy frames (red and yellow)

However, when this solution has been tested with the same video sequence used in the previous section, the results are far from good, being worse than the median filter and even the input image in some cases. [Figure 18]

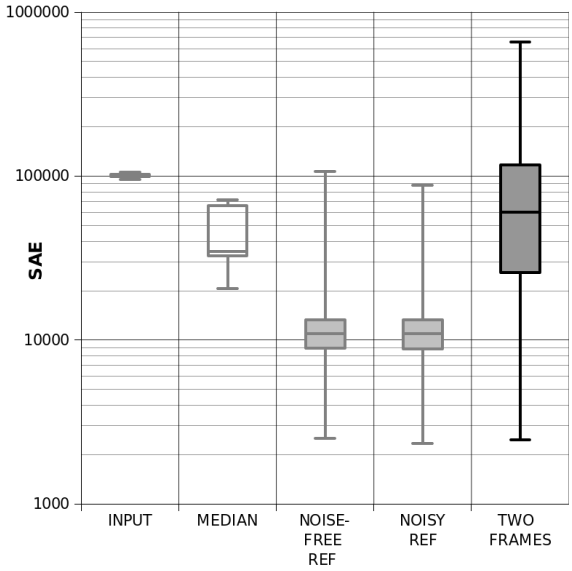


Figure 18. Box plot comparing the results of this approach with the ones obtained in previous experiments

This is a symptom of the filter not being able to converge most of the times because the training input and reference images are too different. One of the factors causing this difference is the fact that the image moves from one frame and the next one, thus causing the evolutionary algorithm to generate a filter that displaces the image in the same way, obtaining high SAE values when comparing the result with the original image despite them being visually similar.

The discrepancy between these experimental results and the ones in Figure 17 is probably due to the fact that when a picture was taken, the photographed people or objects moved as little as possible, but this is not the case for the video sequence used, which cannot be controlled.

Therefore, this approach cannot be used unless it is possible to ensure that the image being captured changes as little as possible between frames.

5.2. Second approach: selection of frames based on similitude

Although many of the frames in the video sequence are too different from the next one to be used as training input and reference images, there are certain couples of

frames that are similar enough. [Figure 19] A way to overcome the problem described before is to search for these frames and perform the evolution with them rather than picking the first two frames available. Thus, before the evolution starts, the system captures a certain number of frames, compares each frame with the previous one in order to find the two most similar consecutive frames, and selects these two frames for the evolution.

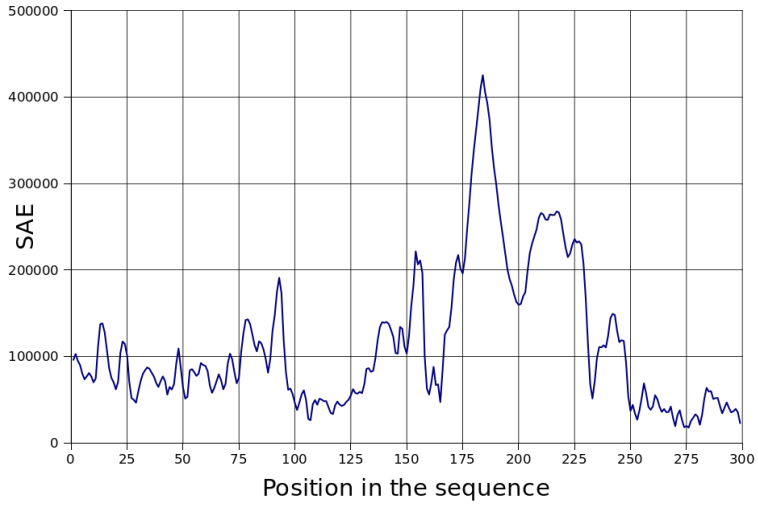


Figure 19. Plot showing the SAE between frames during the video sequence. It can be seen that certain points are exceptionally high compared with their neighborhood.

In order to be able to apply the solution proposed in section 1, these two frames should represent two noisy versions of the same image, or at least two similar enough images. This would require comparing the images without noise in order to evaluate how similar they are. This is not possible since the noise-free images are not available. However, this is not a problem, since as can be seen in Figure 20, the SAE between clean frames and between noisy frames is strongly correlated, thus allowing comparing the noisy frames instead of the noise-free frames in order to find out which frames would be most similar if they were noise free.

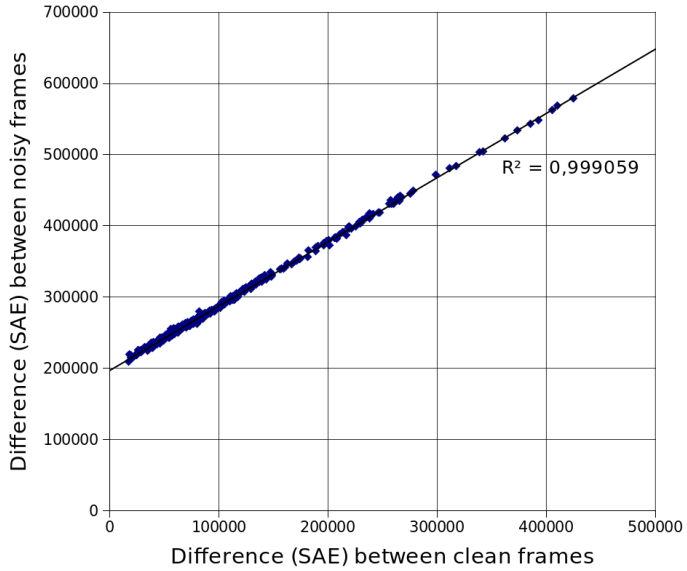


Figure 20. Correlation between the SAE of two noise-free frames and the SAE of the same frames with noise

Another issue that must be addressed is how to compare the frames. This comparison could be performed in software, but this operation would be too slow, so implementing the comparator in hardware would be a good solution.

However, if SAE is used as the measure of similitude between frames, there is no need to implement a dedicated hardware for calculating this: the configurable filter already implements a SAE calculator in hardware. This calculator computes the SAE between the output of the filter and a reference image, but can easily be used to directly compare two images by setting them as input and reference and configuring the filter as a *pass-through filter* whose output is identical to the input. Therefore, no hardware modification is required in order to implement this feature.

Figure 21 shows the results obtained with this method comparing them with the previously obtained ones. As can be seen, a sample of 30 to 60 frames, which is equivalent to 1 or 2 seconds of video at 30 frames per second, produces acceptable results.

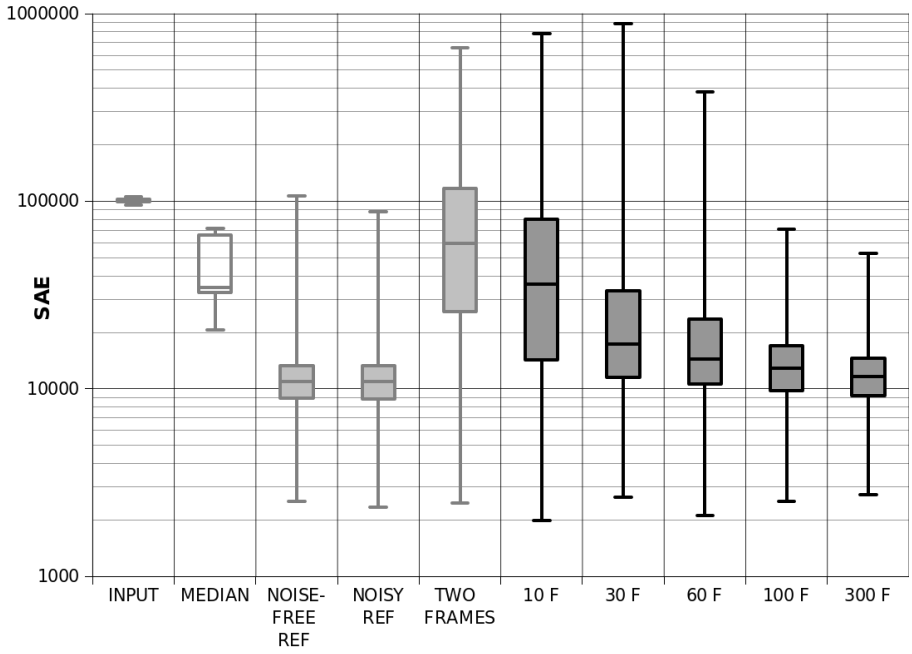


Figure 21. Box plot showing the result of performing the evolution with the most similar frames in a sample of 10, 30, 60, 100 frames, and the entire video sample

Note that both approaches rely on the randomness of the noise, which makes the noise on the two images not to be correlated. These approaches would not work with noise that is correlated on both images, such as defective pixels in the camera sensor.

III. DEVELOPMENT

6. *Modifications to the original implementation*

The proof-of-concept implementation discussed in section 2.3 only showed a small hint of the versatility of evolvable hardware, and has been extended during this work in order to implement new features that would demonstrate its potential.

6.1. Adaptation to a real-time task: the camera

The previously developed implementation was only able to process static images stored in a CompactFlash card. However, this is not enough to demonstrate the real-time capabilities of the system. For this reason, the system has been equipped with an actual input device: a Toshiba TCM8230MD camera.

This camera has already been used at CEI for different projects, with a custom printed circuit board developed at CEI [Figure 22] in order to connect it to different Altera and Xilinx development boards.

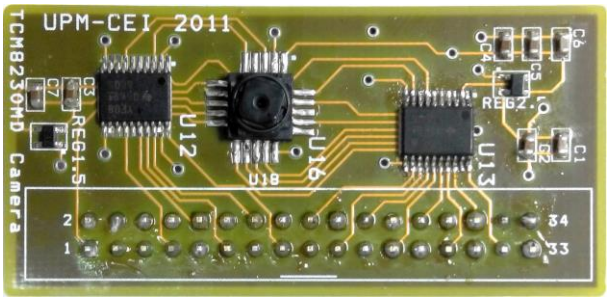


Figure 22. Camera board developed at CEI, featuring the Toshiba TCM8230MD camera

The camera is controlled by a custom peripheral configurable via commands from the MicroBlaze.

Camera controller

The camera controller reads the data from the camera and stores it on a BRAM memory, from where it will later be read by the MicroBlaze. The data generated by the camera have the following characteristics:

- The camera generates an 8-bit parallel signal with image data, together with horizontal sync (HD), vertical sync (VD), and a clock signal (25 MHz with the selected settings).
- Data is valid when the HD signal is high and the clock signal has a rising edge.
- Image data is encoded as YUV data in a U1-Y1-V2-Y2 sequence. Since the application will only use grayscale (Y) data, one out of two bytes has to be discarded.
- The first frame the camera generates after power-up contains garbage data and has to be discarded.

The controller will wait for a command from the MicroBlaze to capture an individual frame. Once this signal is received, it will wait for the camera to start a new frame. The MicroBlaze will periodically poll the controller for a finish signal, which the controller will generate once enough data from the frame has been read and saved to the BRAM memory.

The camera controller can be parameterized to select a rectangle from the data read and discard the rest of the pixels. This will make it easy to crop a 128×128 pixel image from a 176×144 (QCIF) frame.

A problem with BRAM memories is that they only have two ports, so if the controller writes directly to the BRAM for the input image for the filter, and the filter reads from this BRAM, the MicroBlaze will not be able to access said BRAM because it would need a third port. This could be a problem if feedback on the filter performance is needed, but can be easily solved by having two separate BRAMs, and have the MicroBlaze copy the content from one to the other once the capture has finished. The time spent in this memory copying is very short, and also allows the camera to start capturing a new frame without overwriting the old one, which would allow further filtering operations to be performed on it.

It is not convenient to use an external signal as a clock source, but since the camera controller will work at a frequency of 100 MHz, 4 times faster than said clock, the clock can be sampled as a regular signal, and the rising edge condition can be detected by comparing two consecutive samples.

The camera on the proof-of-concept setup could accidentally disconnect and require manual reconnection, after which it would be in a misconfigured state in which it would not generate frames. Therefore, the software would get stuck

indefinitely waiting for a new frame. To avoid this, rather than indefinitely polling for a finish signal, the software implements a timeout after which it will send the reset sequence to the camera and try to capture a new frame.

I²C camera port controller

The Toshiba TCM8230MD camera needs to be configured in order to set the possible options it has: resolution, data format, contrast, brightness... In order to do this, the camera is equipped with an *inter-integrated circuit* (I²C) bus. The camera needs to be configured through this bus before it becomes operative.

The component library on Xilinx Platform Studio includes an I²C controller, the XPS_IIC [24]. This controller would have been a very good option for the system since it would have saved all the work to implement a custom controller. However, this controller did not work.

The camera board developed at CEI uses two Texas Instruments TXB0108 bi-directional voltage-level translators in order to convert between the 3.3 V signals on the FPGA to the 2.8 V signals on the camera, both for the video data and for the I²C controller. However, the datasheet for these integrated circuits [25] explicitly says that "*the TXB0108 should not be used in applications such as I²C or 1-Wire where an open-drain driver is connected on the bidirectional data I/O*". This is because the I²C protocol waits for an acknowledgement at the end of each byte sent by setting the line to a pull-up state and waiting for a logical 0 [26], but these integrated circuits do not handle this pull-up correctly. As a result, the XPS_IIC controller fails right after sending the first byte, aborting the transmission.

This problem had not been noticed before because the only application in which these boards had been used before were on a stand-alone system with no processor in which the I²C controller had been written from scratch, and the acknowledgement was simply ignored.

Replacing the integrated circuits with ones from the TXS01xx series, as suggested by the TXB0108 datasheet, would have involved ordering them, desoldering the old ones and soldering the new ones, which would have implied a long development time wasted, so instead of this, the previously described I²C controller that ignored the acknowledgement was included in the project.

However, this controller was hard-coded to send a predefined sequence at startup, and was hard to extend with new commands, so it eventually had to be rewritten in order to take arbitrary commands from the MicroBlaze and send them to the camera. This way, the driver is not only able to send an arbitrary startup sequence to the

camera, but it also can send commands such as brightness and contrast settings in the middle of operation.

Noise injector

In order to demonstrate the noise filtering capabilities in real time, noise needs to be added to the camera data. This cannot be done offline as it has been done with training images, so the noise has to be generated by the system itself. This is a resource-consuming task that should better be performed by hardware than software.

For this reason, the camera controller has been equipped with a pseudo-random noise generator. This generator is inserted in the camera controller between the data capture and the BRAM controller, so that it adds noise directly to the pixel stream.

This noise injector is based on a 35-bit *linear feedback shift register* (LFSR) as a pseudo-random number generator, modified to generate 32 bits per clock cycle. The noise models this injector can generate are:

- No noise: pixels on the output have the unmodified values of the pixels on the input.
- Salt and pepper noise: when the value of the 32 bits obtained from the LFSR is below a certain threshold value (specified by the MicroBlaze), the pixel is replaced with a black (0) or white (255) pixel. The color is chosen using the least significant bit on the LFSR.
- Impulse noise: when the value of the LFSR is below a certain threshold value, the pixel is replaced with a random color from 0 to 255. The color is chosen using the 8 least significant bits on the LFSR.
- Impulse burst noise: when the value of the LFSR is below a certain threshold value, the output is deactivated for a random amount of pixels, time during which the output will be replaced by white (255) pixels. Both the threshold and maximum deactivation duration are set as parameters by the MicroBlaze.
- White additive noise: all pixel values are incremented or decremented by a random value with uniform distribution, and saturated to 0 or 255 in case of an overflow. This value is obtained by masking the LFSR value with a parameterizable mask in order to be able to select the range of the value; for example, a mask of 31 (a number with the last 5 bits set) would generate values from 0 to 31. Filtering this type of noise is out of the scope of this project, but was implemented as a reference.

6.2. Real-time visualization and interactivity: the demo

So far, the system only allowed evolving with a predefined training image and eventually yielding a result. Although this would be enough for an actual implementation and for testing the system performance, it gives little information on what is actually going on in the system. It is easier to explain what the system does with a real time visualization of its state. Additionally, it would be interesting to see in real time how the system reacts to changes such as variations in the mutation rate or sudden faults on processing elements.

For these reasons, a visual interactive demo was created. Figure 23 shows a screen capture of this demo, displaying the training input image (left), a representation of the filter (middle-left), the output image (middle-right) with the fitness (SAE) value written below it, the training reference image (right), and a plot of the value of the fitness during the evolution (bottom) with smaller values representing better filters.

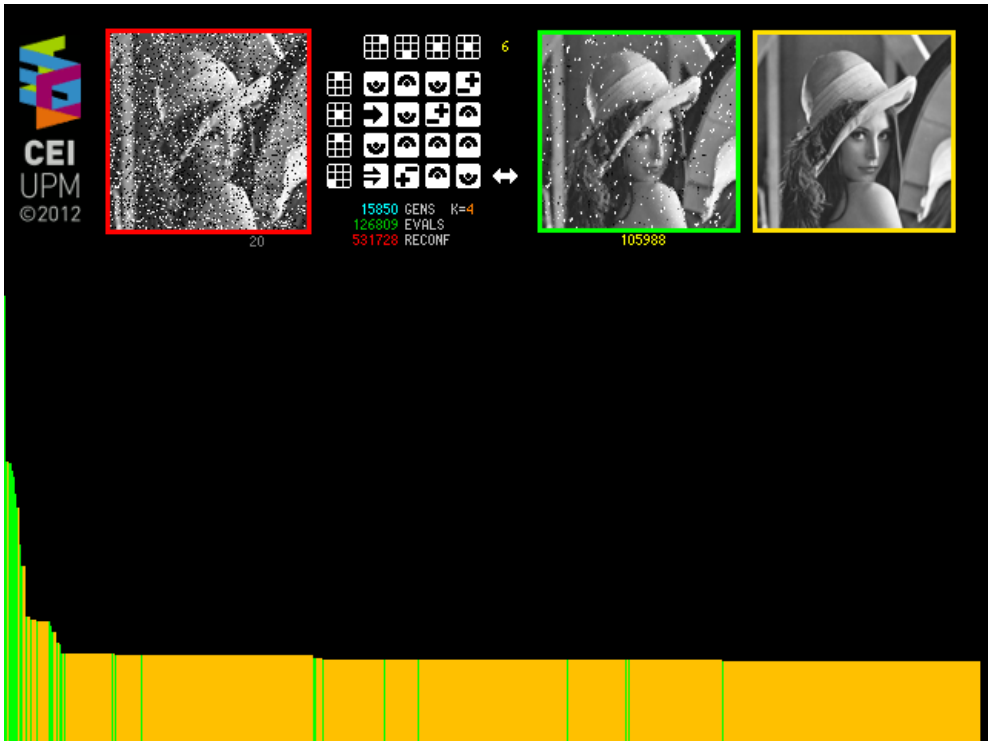


Figure 23. Screenshot of the demo

The demo attempts to show all the features of the system in an interactive manner. It has been elaborated using the XUPV5 board DVI output together with the DVI

controller provided by Xilinx, the XPS_TFT IP core. This controller serves as a back-end that simplifies the task of implementing a 640×480 pixel video output, by mapping a memory region of the DDR2 RAM memory to pixels on the output.

A library of functions to operate this video interface has been developed. This library maps the video memory as a bidimensional array of 32-bit integers, each representing a pixel in the video memory. Functions have been developed to display images, numbers, and other visual elements such as bars and frames.

In addition to the visual feedback of the state of the FPGA, the demo is interactive and can be manually operated. This is achieved through a PS/2 keyboard attached to the board, using the corresponding controller supplied by Xilinx. The keyboard is polled periodically to check for key presses. However, the keyboard is not needed for the demo to run, and can be used without it. The keyboard can be hot-plugged during the demo.

Certain keys on the keyboard are mapped to specific functions on the demo. In order to easily remember which key maps to which function, a keyboard was customized to represent the different functions. [Figure 24]

Figure 24. Customized keyboard used in the demo

The demo also includes the camera controller in order to visually demonstrate the real-time capabilities of the system.

The features implemented in the demo include:

- Pause, continue, and re-start the evolution
- Re-start the evolution with a different pair of images, in order to demonstrate the adaptivity capabilities of the system. There are 9 preset images that can be changed by just replacing the images stored in the CompactFlash card. The preset images are:

  - Input with 20% of noise, noise-free reference; with salt and pepper, impulse, white additive, and impulse burst noise.
  - Same of the above, but with a noisy reference with the same type and level of noise as the input.
  - Noise-free input, and image processed with an edge detection filter as reference. This demonstrates that the system is also able to adapt to problems different from filtering noise, by generating an edge detection filter with only an example of what the result should look like.
- Dynamically set the mutation rate
- Simulate permanent faults in PEs
- Simulate a cascaded filter with 3 identical filters, by copying the filter output to the input 3 times
- Put the demo in real-time mode, halting the evolution process and filtering images captured from the camera with the corresponding noise type and level set, displaying said images in real time
- Set the brightness, contrast, and noise type and level of the camera
- Capture two consecutive frames and perform the evolution using them as training input and reference
- Save a screenshot of the video memory to the CompactFlash card

Figure 25 shows some screenshots of the demo displaying some of these features.

This demo has been very useful for showing what our current project and research lines were about to people interested on them, and has been displayed in demo sessions and exhibits at several conferences [27] [28] [29], allowing to spread the usefulness of evolvable hardware and DPR.

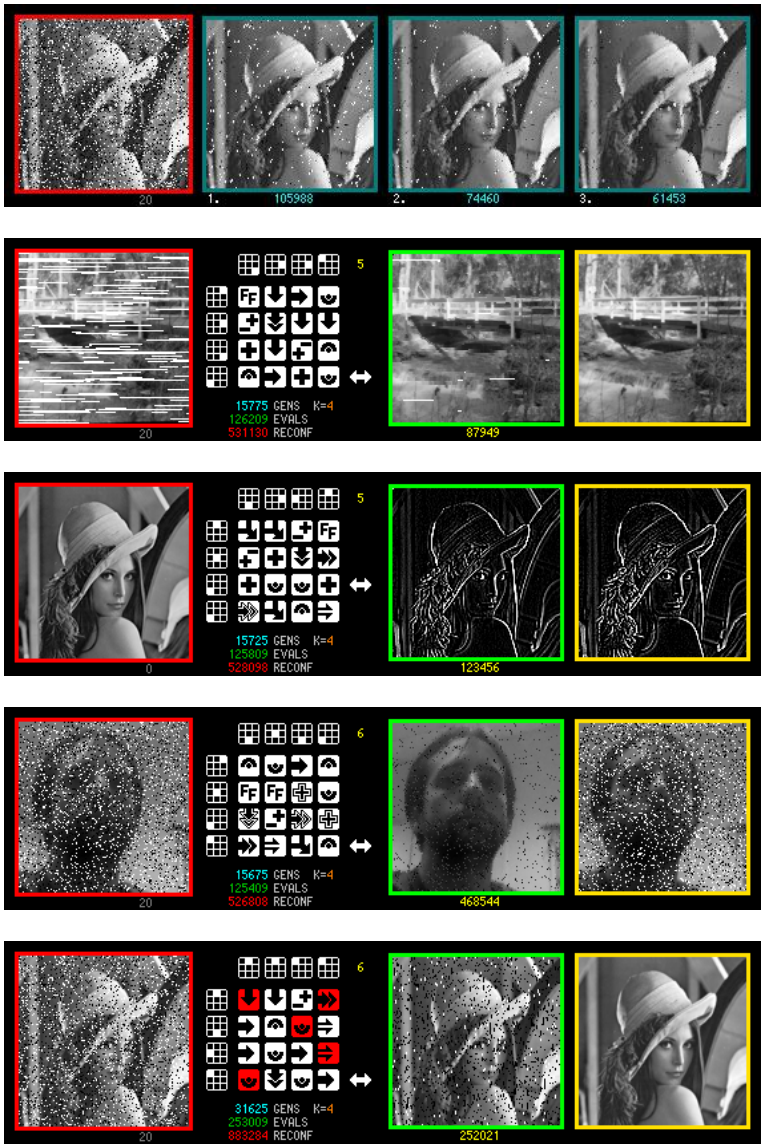


Figure 25. Some other screenshots of the demo, showing cascaded filtering, burst noise, edge detection, evolution with captured images, and fault emulation.

6.3. Other improvements

6.3.1. Smaller processing elements

The region of the FPGA dedicated to reconfiguration that holds the PEs of the systolic array was very large, vertically measuring a 50% of the size of the FPGA. This complicated the task of adding new elements to the system, since the routing task became more complicated and the available space was small. Additionally, it prevented creating a system with multiple systolic arrays, which was a parallel research line.

For these reasons, it has been necessary to modify the size of the processing elements. The new processing elements are 4 times smaller in height, making the size of the systolic array be only a 13% of the total FPGA height.

However, this reduction in size has consequences. The former processing elements measured exactly one FPGA row, which made reconfiguration an easy task. The new ones are one quarter of a row, which, as has been seen in 2.1.4, causes reconfiguration to be done in two steps: reading the configuration memory content, and writing it back with some values changed. This has caused the configuration time to double.

Additionally, the read operation has speed limitations. Formerly, the HWICAP was overclocked at 200 MHz; however, the read operation has limited its speed, causing the HWICAP to fail for such speed. For this reason, the speed has been reduced to 100 MHz.

As a result, although the size of the reconfigurable zone has been reduced to a 25% of the original, the reconfiguration time has grown to 4 times the initial one. This reduces the speed of the evolutionary algorithm, which is now about 2.4 times slower.

Solving this problem is out of the scope of the current project, and will be done in a future.

6.3.2. Random number generator

The random number generator for the evolutionary algorithm was based on a 32-bit *linear congruential generator* (LCG) implemented in software. This random value was later shifted 16 bits in order to discard low randomness bits, and the rest modulo n was calculated in order to obtain an integer between 0 and n–1:

```
uint32_t seed = 1;
int rand_n(int n) {
    seed = seed * 1103515245 + 12345;
    return (seed >> 16) % n;
}
```

Using this random number generator, the system was able to randomly mutate a genotype in about 23 microseconds. This time was comparable to the reconfiguration time (70 µs) and filter time (82 µs), but since this process was parallelized with the filtering process, it barely affected the total time.

In order to make this function generate better random numbers, this code was later changed to

```
return seed / (0xFFFFFFFF/n + 1);
```

However, as new evolutionary algorithms were tested, it was noticed that this code significantly slowed them down. The reason of this is that, while the generation of the 32-bit random number takes around 0.15 µs (or up to 0.70 if the standard `rand()` function is used instead of this LCG), the computation of a modulo or division operation takes about 4 µs. This is because the MicroBlaze processor is not equipped with a hardware divider unless this option is explicitly enabled, and thus the division operation is performed as a software routine. [19]

This problem has been solved by modifying the function in this way:

```
return ((seed >> 8) * n) >> 24;
```

This solution reduces the maximum value of n to 256, but since it only involves shift operations and a multiplication, both available as MicroBlaze instructions, it is performed considerably faster than the previous versions without having to modify the hardware in order to implement a divider.

7. Further development

In addition to the improvements made in the system, some software tools have been developed which complement the work: a *software emulation of the system* and a *bitstream manipulation tool*.

7.1. Software emulation of the system and demo

In order to test the hardware implementation and make design decisions without having to implement the hardware, a software emulation of the system that runs in a desktop computer has been implemented.

This emulation has been programmed in C, which is the same language used in the embedded software that runs on the FPGA, and thus makes the code compatible between both platforms.

This is not the first time a software-based emulation of this system is developed: a software emulation of the system is already presented in [1]. This emulation had been programmed in Cython [30], a Python-based tool that allows easily implementing compiled Python modules integrating C source files for functions that should run fast. This was used to build a Python module which would later be used to develop genetic algorithms in the form of Python scripts.

However, this Cython implementation had a problem: if an evolutionary algorithm is developed in the form of a Python script, it will have to be translated to C in order to implement it on the actual embedded system running on the FPGA. For the same reason, algorithms designed in C for the FPGA would not work in the emulation. In order to overcome this problem, it was decided to drop the Cython implementation of the emulated system and rewrite it in pure C.

Although this may seem like a complicated task, most of it has been carried out by reutilizing code or using specific libraries:

- The emulation of the systolic array had already been implemented as a C function for the Cython implementation.
- Platform-specific functions such as `sysace_fopen()` and `sysace_fwrite()` are almost equivalent to standard C functions such as `fopen()` and `fwrite()`, so it is trivial to implement the former in terms of the latter.
- The *Simple DirectMedia Layer* (SDL) library [31] is a cross-platform multimedia library that, among other things, allows creating video framebuffers. The format

of these framebuffers is (or can be made) identical to the one used by the XPS_TFT video controller. For this reason, the only requirement to port the graphic functions from the FPGA implementation to the SDL one is to create a few setup and periodical refresh functions. Additionally, SDL allows keyboard interaction, which makes porting the keyboard control functions to the emulated system an easy task.

- The rest of the software, including algorithms and helper functions, was already implemented for the FPGA, and only needs minor tweaks in order to be adapted to the emulated system.
- The camera is not implemented on the emulation.

This emulated version of the system allows testing different system variations that would otherwise involve resynthesizing the hardware, such as changes on the PE functions or the fitness calculation function. Additionally, it allows making video captures of the demo using a simple *screencap* software tool on a desktop computer, while doing this directly with the video output on the FPGA board would be very complicated.

The emulated demo is 5 times slower than the hardware system and generates exactly the same results. This is not a huge difference, due among other things to the fact that the HWICAP had to be slowed down as explained in section 6.3.1.

7.2. Partial bitstream extraction tool

Partial bitstreams used for the PEs within the systolic array were generated by synthesizing a circuit and extracting the content of a region with a partial bitstream extraction tool. This tool was programmed in C, which does not provide many tools for easy string and list manipulation, making the source code complicated, and would require rewriting it in case new FPGA models had to be added.

Additionally, the sequential extraction of multiple partial bitstreams was performed through a Bash [32] script, using the Bash interpreter included in the Xilinx suite. Unfortunately, newer versions of Xilinx for Windows do not include this interpreter anymore. [33]

For these reasons, it was decided to rewrite the partial bitstream extraction tool in a higher level language: Python. This language allows implementing the extraction tool in a simpler and more readable way using features of object-oriented programming, and extending it by just including additional easy to write Python scripts containing only basic information of the target FPGA, rather than rewriting the whole program.

Additionally, the new bitstream tool allows analyzing bitstream files to extract information, and it has some other utilities such as a bitstream parsing tool that allows reading and displaying the different commands that are sent to the ICAP before the reconfiguration starts. This may be useful in future development when analyzing these commands for a new FPGA implementation is needed.

The resulting Python tool can be used within Python scripts that could serve as an alternative to the Bash language. Additionally, it can be used for easily creating interactive shells or even basic GUI applications. [Figure 26]

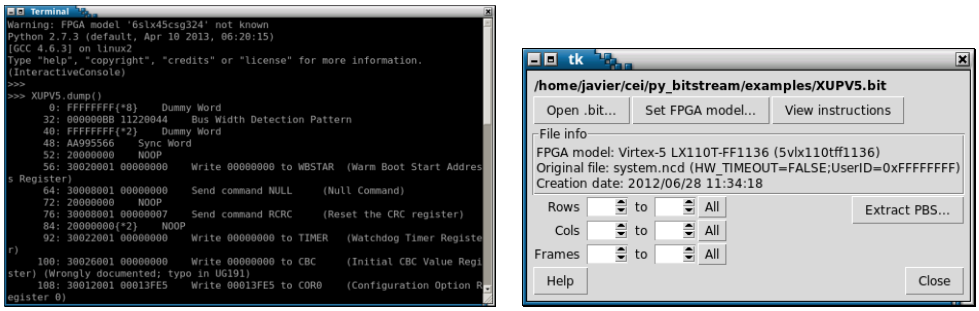


Figure 26. Screenshots of an interactive shell (left) and a GUI application (right) based on the new partial bitstream tool

IV. EXPERIMENTAL RESULTS

8. Experimental results and assessment

In order to evaluate the solutions proposed in sections 4 and 5, several batches of tests have been launched.

Since the aim of the project is to be able to filter video in real time, these tests use an actual video sequence for training and testing the results. The chosen video sequence is a grayscale version of the *Foreman* sequence [34], a publicly available 300 frame sequence often used in image processing.

This sequence has been scaled down to 128×128 pixels, and a 5% level of salt and pepper noise has been added to each of the frames.

The tests performed use the evolutionary algorithm described in 2.3.3, using a mutation rate of $K = 3$. In order to obtain a single filter, 5 evolutions of 20 000 generations are performed, and the best result of the 5 evolutions (that with the smallest SAE) is chosen as the final filter. This filter is then tested with the 300 images of the sequence, using noiseless frames as reference, thus obtaining 300 SAE values that characterize the filter performance. Each experiment generates and characterizes 300 filters, each one obtained from each of the 300 frames of the sequence (or starting on that frame in the last case), thus obtaining a total of 90 000 SAE values per experiment.

The experiments performed are:

- Evolution with a noise-free training reference, which is the method that was used in [1] and related work and will be used as a comparative reference
- Evolution in which both the input and reference images are noisy versions of the same image, as proposed in section 4
- Evolution in which training input and reference images are two consecutive noisy frames, as proposed in 5.1
- Evolution in which training input and reference images are two consecutive noisy frames, previously selected from a sample of 10, 30, 60, 100, and 300 frames based on their similitude, as explained in 5.2

The amount of time used in performing the 5 evolutions needed to obtain a single filter is of 4 minutes.

8.1. Box plot

Figure 27 shows a box plot (anticipated in section 5) displaying these 90 000 results for each of the experiments, together with box plots for the 300 SAE discrepancies of the input images and the ones that would be obtained by filtering the image with a median filter that uses a 3×3 pixel window.

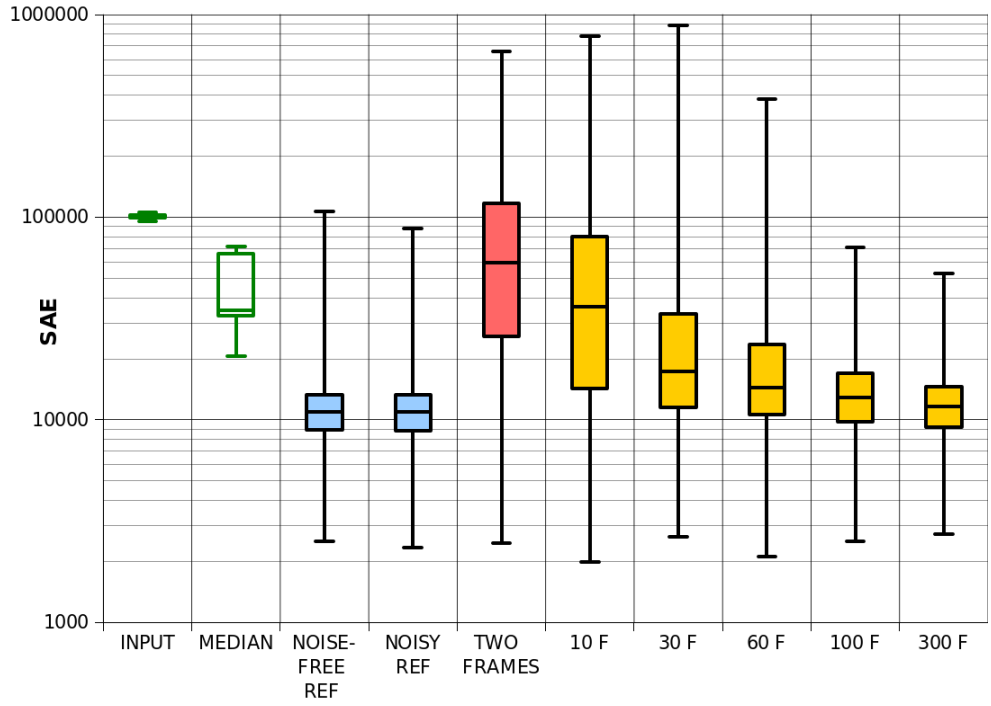


Figure 27. Box plot showing the median and dispersion of SAE discrepancies with the original at the input, after filtering the image with a median filter, and after filtering the input image with a filter trained using each of the discussed methods

As can be seen, training with a noisy reference leads to almost identical results than the ones obtained by training with a noiseless reference, therefore proving the validity of this method. Both results outperform the median filter by a clear difference. This means that using two noisy images as training input and reference is equally valid as an evolution method; however, two noisy versions of the same image will rarely be available.

Training with two consecutive frames selected from a sample of 2 or 10 frames yields very bad results, in many cases worse than the ones obtained with the median filter and even worse than the input images in some cases. These results are unacceptable.

Training with two consecutive frames selected from a sample of 30 or 60 frames, which is equivalent to a video sample of 1 to 2 seconds at 30 frames per second, yields much better results. This method outperforms the median filter results in most cases. There are still, however, some SAE values that are remarkably high; nevertheless, these are atypical values that can be ignored (these values could have been displayed as outliers; however this would have resulted as a cloud of points not giving any useful information).

A problem with box plots is that they are not very good at showing outliers for a high amount of points; information of the first or fifth percentiles would be more useful.

Training with two consecutive frames selected from a sample of 100 (3.3 seconds) or 300 frames (all the video sequence) yields results almost as good as the ones obtained using a noiseless reference. However, especially in the case of the 300 frames, the reduced data dispersion is partly due to the fact that all the filters will have chosen the same pair of frames for the evolution rather than several different pairs, so these results may not be statistically reliable.

In conclusion, a sample of 60 frames represents a good compromise choice between sample size and obtained results for this video sequence. This solution implies wasting only 2 seconds, which is a small amount of time compared with the one spent by the evolutionary process. Alternatively, frame acquisition could be performed in parallel while an evolution is being performed, therefore reducing the overall time.

Notice that these results may vary when different video sequences with different conditions such as frame rate or amount of motion are used.

8.2. Mean-stdev plots

A problem with these box plots is that they do not show characteristics of each of the obtained filters, but only an overall result of all the SAE values mixed.

In order to better characterize the properties of the filters, the scatter plots in Figure 28 have been generated. Each point in the plot represents the mean and standard deviation of the logarithms of the SAE values obtained with a filter when the 300

frames are filtered using it. Additionally, the median filter and the mean and standard deviation of the input images are included as references.

These plots provide an easy way to assess the overall behavior of the filters obtained with each method.

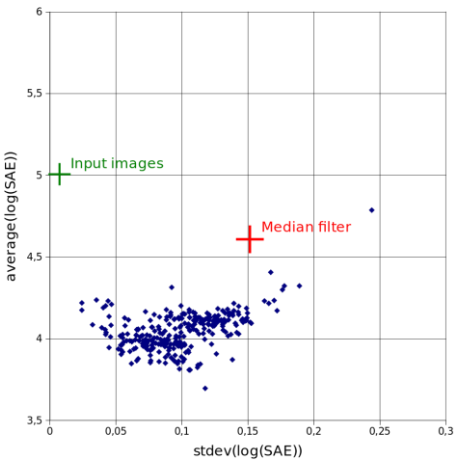
As can be seen, both the evolution with a noiseless reference and with a noisy version of the same image yield better results than the median filter: apart from some exceptions, both have better performance (mean) and predictability (standard deviation) than the median filter.

When two consecutive frames are used, however, a densely populated protuberance appears on the top of the chart. This protuberance grows further away from the value of the mean for both the median filter and the input SAE. This indicates that the filters created by this method often yield bad results.

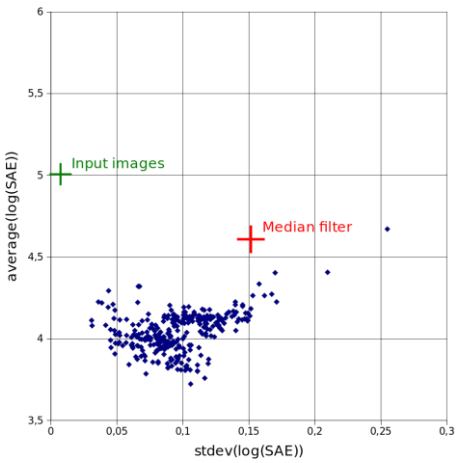
As more samples are used for selecting the most similar consecutive frames, this protuberance becomes less densely populated; nearly disappearing with 60 frames and vanishing with 100.

With 300 frames, not only the protuberance is gone, but the results have an even smaller standard deviation than the ones of evolving with a noiseless reference. This is due to the fact that these filters have been created with the frame in the sequence with least motion, which will be the most suited for the evolution.

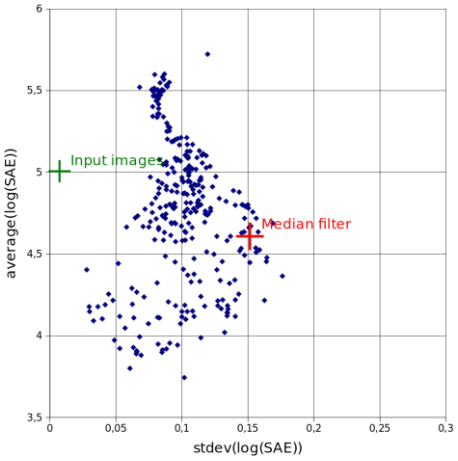
These plots confirm the previously obtained conclusion: a sample length of 60 frames seems to be the most convenient for this video sequence.


Input images
Median filter
average(log(SAE))
stdev(log(SAE))

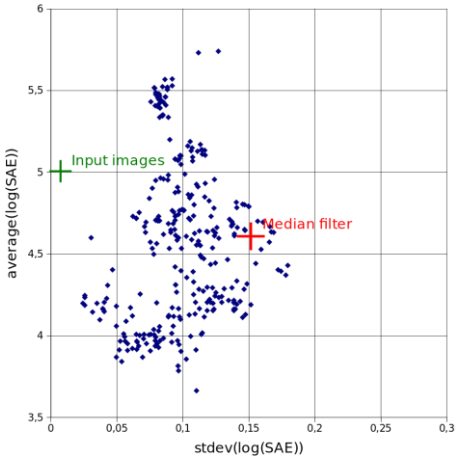
(a)


Input images
Median filter
average(log(SAE))
stdev(log(SAE))

(b)


Input images
Median filter
average(log(SAE))
stdev(log(SAE))

(c)


Input images
Median filter
average(log(SAE))
stdev(log(SAE))

(d)

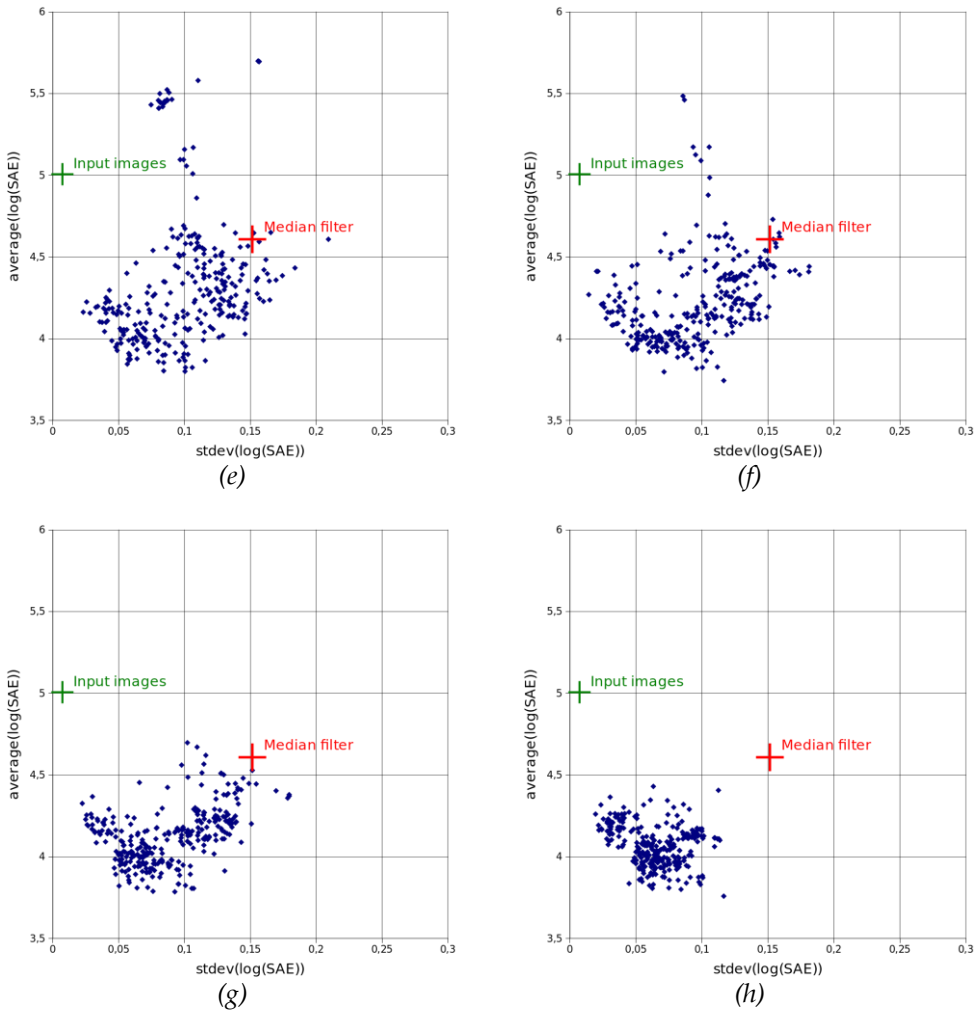


Figure 28. Mean (vertical) vs. standard deviation (horizontal) scatter plots for the evolution with a noiseless reference (a), a noisy reference (b), two consecutive frames (c), and two consecutive frames selected from a sample of 10 (d), 30 (e), 60 (f), 100 (g), and 300 frames (h).

8.3. Other noise types and levels

In order to prove the adaptivity of the solution, another batch of experiments has been performed. These experiments use the strategy discussed above: the two most

similar consecutive frames in a sample are used as training input and reference for the evolution. The sample length is 60 frames, which has proven to be a good choice.

The noise types tested are salt and pepper noise with a level of 5% (same as previous experiments), 10%, and 20%; and impulse and impulse burst noises with a level of 5%.

Each experiment generates 50 filters using the same evolution strategy (best of 5 attempts of 20 000 generations). For each of them, the sample from where the frames are picked starts at a random point in the sequence. Note that this is a smaller amount of results than the 300 filters that were generated in previous experiments.

Figure 29 shows a box plot of each of the experiments (gray with black outline), comparing them with the results obtained with the median filter for the same type of noise (white with gray outline).

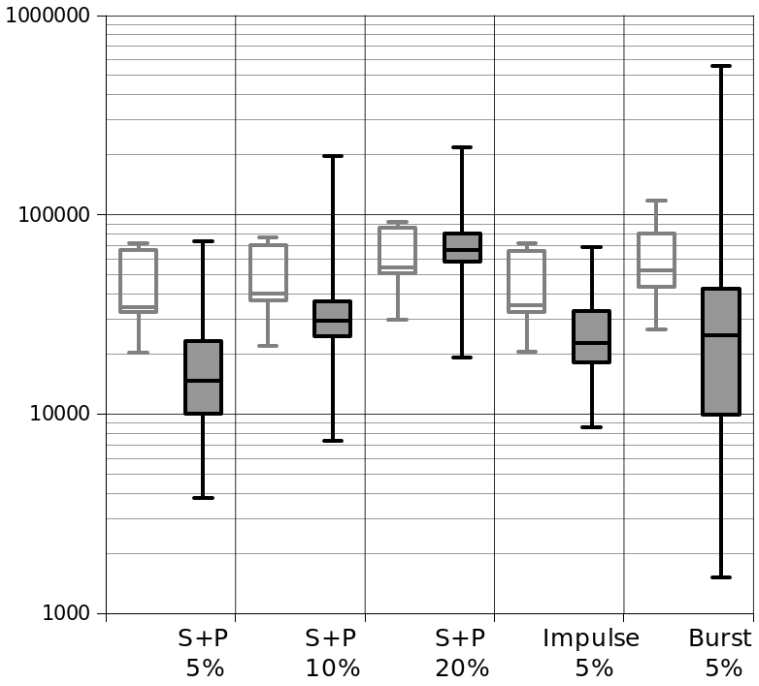


Figure 29. Comparison of the performance of the evolved filters with different types and levels of noise in the images. From left to right: salt and pepper noise with levels of 5, 10 and 20%; impulse noise at 5%; impulse burst noise at 5%.

As can be seen, as the noise level increases, this system loses efficiency over the median filter, reaching similar values with a level of 20%. As for impulse and impulse burst noises with a 5% level, the results are also better than the ones

obtained by the median filter; however, there is noticeable data dispersion for the impulse burst noise.

Figure 30 shows some of the visual results that were obtained during these tests, with rows (1) to (5) representing 5%, 10% and 20% salt and pepper noise, 5% impulse noise, and 5% impulse burst noise respectively. Column (a) is the input image using for filtering; (b) is the result obtained when filtering that image after the evolution has finished; (c) and (d) are the input and result of filtering another frame in the sequence with the filter trained with (a).

As an overall conclusion, the proposed strategy outperforms the median filter in most cases. It has also been found out that two consecutive frames cannot be directly used as input and reference images, at least on video sequences with motion levels similar to the one used for the experiments; nevertheless, this can be solved by searching the most similar frames in a short sample.

Figure 30. Some of the obtained results

V. CONCLUSIONS AND FUTURE WORK

9. Conclusions

Results of the work

In this work, the adaptivity capabilities of a previously designed system have been tested. It has been shown that the system, which had only been tested for a specific type of noise (salt and pepper noise) can adapt to other types of noise and some other filtering tasks without modifying the hardware, by just supplying the appropriate training images.

Furthermore, a method for filtering several types of noise without having to identify or model it has been shown and tested with a real video sequence. This method has not required the development of any additional hardware for this specific task: the hardware used to compare frames in search for two similar ones is the same used to calculate the fitness of a filter.

In addition, a real-time application of the system has been developed; and an interactive implementation with real time display of the results, which were set as two of the future objectives in [1], has been created. This interactive demo, although may not be of direct research interest, allows people interested in the topic understanding how an evolvable hardware and dynamically reconfigurable system works. Therefore, this is by itself an important contribution of this work.

Possible applications

A system like this could be beneficial in applications that need to be autonomous and hardware-accelerated. For example, a video device that records and compresses images before transmitting or storing them may require a pre-processing filtering of those images. It may not be possible to perform this filtering process after the compression, so the process would have to be carried out by the device. There is a wide range of applications with such requirements, from meteorological satellites and space probes to video camera recorders and surveillance systems.

Another application of this evolutionary filter would be the hardware acceleration of filtering tasks. This system is not only beneficial for autonomous applications:

intrinsic evolution is a desirable feature for its speed and the lack of possible discrepancies between a simulation model and the real system.

The proposed training method with no noise-free references is also interesting regardless of whether the system is autonomous or not. Although similar approaches had already been proposed in [9] and [10], its application to evolvable hardware is a novel idea.

Publications

This project, including this and other related works, has led to the publication of several papers and journal articles. [35] [22] [36] [37] [38]

10. Current and future research lines

This work has solved some of the issues of evolvable hardware, but there is still a lot of research work and improvements to be made on this topic.

10.1. Current new research lines

Part of this research work is already being developed at CEI, and is the topic of several bachelor, master, and PhD theses:

Scalability

Certain problems may require a processing core of a larger complexity than the one a 4×4 systolic array can provide. For this reason, a scalable array is being developed which allows dynamically changing the amount of resources used, shrinking or enlarging the array. [38] This new system makes a further usage of DPR, also replacing the input and output multiplexors with reconfigurable elements, so that all the reconfiguration is performed via DPR.

Additionally, parallel or cascaded filtering can be performed by introducing a variable number of arrays. [37]

Power consumption

Some filter configurations consume more power than others, especially if such configurations use more resources due to the aforementioned scalability. In some situations, such as low power applications, a tradeoff between filtering quality and power consumption may be desirable. For this reason, research in power-aware evolution strategies is being carried out, focused on developing a multi-objective evolutionary algorithm that considers both filter performance and power consumption.

Partial bitstream design

Right now, design of reconfigurable circuits involves placing special circuits known as *bus macros* [14] in order to keep the partial circuit inputs and outputs aligned with those of the static design. Additionally, the developer has to visually check that no nets cross the boundaries of reconfigurable regions, manually routing them in that case.

In order to overcome these problems, a tool for automating the routing task is being developed, which will also handle the input and output alignment, thus eliminating the need for bus macros. [39]

10.2. Possible improvements and future research lines

Apart from the aforementioned research lines, improvements could be done to the system to boost its speed and performance features.

Filter operation

There are some issues that prevent the system for being fully suited for autonomous real time applications.

First, although the system adapts to the input conditions autonomously before entering in mission mode, it is not yet able to decide when the input conditions have changed and it should enter training mode again. This could be addressed by measuring the filter action (how much the input and output differ) and monitoring if it changes, or by periodically re-launching the training stage regardless of the input conditions.

Second, a real time evolution method that may yield better results than using the same couple of frames for the five attempts used to obtain a single filter would be to change the samples used in each of these attempts. This would be better than using the same couple of frames for all of them, since the sample size would be increased by 5 times and filters resulting from evolving with a bad couple would likely be discarded; however, it would be worse than simply making the sample 5 times longer.

Finally, although the time taken to train the filter is quite short, the filter remains inoperative during this time. A better approach would be to interleave evolution and filtering, so that after a frame has been filtered and before the next frame is available, a few generations of the evolution can be carried out. This way, the filter keeps working while it gets recalibrated, thus making it more suited for real time applications.

Evolution strategies

As new features such as scalability are added to the system, new evolutionary algorithms handling these features have to be designed.

In the case of scalability, the algorithm not only has to improve a fixed size array, it also has to determine when the array should grow to cover more complex tasks, or when the task is simple enough and the array can be shrunk.

Additionally, although the topic most exhaustively being researched here is partial reconfiguration of hardware, it is important to study possible evolutionary algorithms that may improve the system performance: making the system evolve in half the evaluations is as beneficial for the training time as making it able to perform these evaluations in half the time. These improvements can go from tweaking evolution parameters such as the mutation rate, the number of children, or the number of attempts to studying different approaches such as variable mutation rates, different fitness evaluations, or other evolutionary strategies such as crossover and tournament.

Hardware improvements

As seen in 6.3.2, modifications in the processing elements have caused the HWICAP to work 4 times slower, reducing the training speed about 2.4 times. This causes the system to lose part of its advantage over software-based solutions such as the one proposed in 7.1.

As has been pointed out, this slowdown is due to the need of reading the FPGA configuration memory, so a solution to this problem would be to avoid doing so. It has been seen that this is due to the impossibility of reconfiguring a region smaller than an FPGA row, thus having to read all the content, replace part of it with the new one, and write it back.

Instead, all the new content could be generated directly by concatenating data from multiple partial bitstreams, thus eliminating the need of the aforementioned readback. This could be done by interleaving multiple burst read operations from different positions of the DDR2 RAM memory, or by using a faster memory that allows more arbitrary data access such as the BRAM, although this memory is a limited resource that may not suffice for this purpose.

Additionally, since this memory is faster, some optimizations may let the HWICAP work at higher speeds, by overclocking the ICAP port to very high frequencies. In [40], ICAP frequencies of up to 550 MHz are achieved for an FPGA of the same family.

Reconfiguration time can also be improved by reducing the size of the processing elements to half the width. This has already been done for the scalable array using the partial bitstream design tool in development.

Another bottleneck may be the peripheral that feeds data to the systolic array. Recent experiments and modifications have shown that the current speed of 200

MHz may be too high when certain modifications are performed, so it would be a good idea to optimize it as well. One of its main bottlenecks may be in the removal of the 1 pixel border of the image: it should be evaluated whether this border really affects the calculation of the fitness and the subsequent evolution results, and it may be a good idea to bypass it, including the border in the fitness calculation.

Additionally, the aforementioned peripheral is hard-coded to operate with a 3×3 window and a 128×128 image. It could be modified so that these values are parameterized at runtime, allowing alternatives such as a 5×5 window and a 176×144 image.

Finally, the usage of the processor could be reduced or even removed by implementing the evolutionary algorithm in hardware, developing a module that communicates directly with the HWICAP and generates genotypes autonomously.

REFERENCES

[1] **Javier Mora** – *Implementación de hardware evolutivo en un array sistólico mediante reconfiguración parcial* (B.Sc. thesis) – Universidad Politécnica de Madrid – Madrid (Spain), 2011

[2] Lukáš Sekanina – *Virtual reconfigurable circuits for real-world applications of evolvable hardware* – Proceedings of the 5th Conference on Evolvable Systems: From Biology to Hardware – Berlin (Germany), 2003

[3] Gonzalo R. Arce – *Nonlinear Signal Processing: A Statistical Approach* – John Wiley & Sons – 2005

[4] Zdeněk Vašíček, Lukáš Sekanina – Novel Hardware Implementation of Adaptive Median Filters – 11th IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS), pp.1,6 – Bratislava (Slovakia), 2008

[5] H. Hwang, R. A. Haddad – Adaptive median filters: new algorithms and results – IEEE Transactions on Image Processing, 4(4), pp.499–502 – 1995

[6] Zhou Wang and David Zhang – Progressive switching median filter for the removal of impulse noise from highly corrupted images – IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing, 46(1), pp.78–80 – 1999

[7] S. Marshall – New direct design method for weighted order statistic filters – IEE Proceedings on Vision, Image and Signal Processing, vol.151, no.1, pp.1–8 – 2004

[8] Raymond H. Chan, Chung-Wa Ho, and Mila Nikolova – Salt-and-pepper noise removal by median-type noise detectors and detail-preserving regularization – IEEE Transactions on Image Processing, 14(10), pp.1479–1485 – 2005

[9] Xiang Zhou, William G. Wee – *Adaptive order statistic filters for noise characterization and suppression without a noise-free reference* – 1998 IEEE International Conference on Communications (ICC), Conference Record, vol.3, pp.1774–1778 – Atlanta (USA), 1998

[10] Mahmoud Saeidi, Seyed Ahmad Motamedi, Alireza Behrad, Behzad Saeidi, Roghayeh Saeidi, Reza Saeidi – *Noise reduction of consecutive images using a*

new adaptive weighted averaging filter – 2005 IEEE Workshop on Signal Processing Systems Design and Implementation (SiPS), pp.455–460 – 2005

[11] Adrian Thompson – *Silicon Evolution* – Proceedings of Genetic Programming 1996 (GP-96), pp.444–452 – Stanford (CA, USA), 1996

[12] Julian F. Miller – *An empirical study of the efficiency of learning boolean functions using a Cartesian Genetic Programming approach* – Proceedings of the 1st Genetic and Evolutionary Computation Conference, vol. 2, pp. 927-936 – San Francisco (CA, USA), 1999

[13] Andres E. Upegui – *Dynamically reconfigurable bio-inspired hardware* (Ph.D. thesis) – École Polytechnique Fédérale de Lausanne – 2006

[14] Davin Lim, Mike Peattie (Xilinx, Inc.) – *Two Flows for Partial Reconfiguration: Module Based or Small Bit Manipulations* [XAPP290] – 2002

[15] Jim Tørresen, Geir Aarstad Senland, Kyrre Glette – Partial Reconfiguration Applied in an On-line Evolvable Pattern Recognition System – 26th NORCHIP Conference, pp.61–64 – Tallinn (Estonia), 2008

[16] Kyrre Glette, Jim Tørresen, Mats Hovin – *Intermediate Level FPGA Reconfiguration for an Online EHW Pattern Recognition System* – 2009 NASA/ESA Conference on Adaptive Hardware and Systems (AHS), pp.19–26 – San Francisco (CA, USA), 2009

[17] Xilinx, Inc. – *Virtex-5 FPGA Configuration User Guide* [UG191] – 2012

[18] Xilinx, Inc. – *LogiCORE IP XPS HWICAP (v5.00a)* [DS586] – 2010

[19] Xilinx, Inc. – *MicroBlaze Processor Reference Guide* [UG081] – 2011

[20] Yana E. Krasteva, Eduardo de la Torre, Teresa Riesgo, Didier Joly – *Virtex II FPGA Bitstream Manipulation: Application to Reconfiguration Control Systems* – 16th International Conference on Field Programmable Logic and Applications (FPL), pp.1–4 – Madrid (Spain), 2006

[21] Andrés Otero, Ángel Morales-Cas, Jorge Portilla, Eduardo de la Torre, Teresa Riesgo – *A Modular Peripheral to Support Self-Reconfiguration in SoCs* – 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools (DSD), pp.88–95 – Lille (France), 2010

[22] Rubén Salvador, Andrés Otero, **Javier Mora**, Eduardo de la Torre, Teresa Riesgo, Lukáš Sekanina – *Self-Reconfigurable Evolvable Hardware System for Adaptive Image Processing* – IEEE Transactions on Computers, vol.62, no.8, pp.1481–1493 – 2013

[23] Rubén Salvador, Andrés Otero, **Javier Mora**, Eduardo de la Torre, Lukáš Sekanina, Teresa Riesgo – *Fault Tolerance Analysis and Self-Healing Strategy*

of Autonomous, Evolvable Hardware Systems – 2011 International Conference on Reconfigurable Computing and FPGAs (ReConFig), pp.164–169 – Cancun (Mexico), 2011

[24] Xilinx, Inc. – *XPS IIC Bus Interface (v2.03a)* [DS606] – 2011

[25] Texas Instruments Inc. – *8-bit bidirectional voltage-level translator with auto-direction sensing and ±15-kV ESD protection* [SCES643E] – 2006/2012

[26] NXP Semiconductors – *I²C-bus specification and user manual* [UM10204] – 2012

[27] Andrés Otero, **Javier Mora**, Rubén Salvador, Ángel Gallego, Eduardo de la Torre, Lukáš Sekanina – *Evolvable hardware FPGA-based platform for autonomous fault-tolerant systems* – 2012 Design, Automation & Test in Europe (DATE), University Booth – Dresden (Germany), 2012

[28] Andrés Otero, **Javier Mora**, Rubén Salvador, Ángel Gallego, Eduardo de la Torre, Lukáš Sekanina – *Evolvable hardware FPGA-based platform for autonomous fault-tolerant systems* – 2012 International Conference on ReConFigurable Computing and FPGAs (ReConFig), Demo Night – Cancun (Mexico), 2012

[29] Ángel Gallego, **Javier Mora**, Andrés Otero, Blanca López, Eduardo de la Torre, Teresa Riesgo – *A Self-Adaptive Image Processing Application Based on Evolvable and Scalable Hardware* – 23rd International Conference on Field Programmable Logic and Applications (FPL), Demo Night Session – Porto (Portugal), 2013

[30] Cython: C extensions for Python – http://cython.org/

[31] Simple DirectMedia Layer – http://www.libsdl.org/

[32] Bash – Gnu Project – Free Software Foundation – http://www.gnu.org/software/bash/

[33] Xilinx, Inc. – *13.1 EDK - How do I migrate my custom make and Tcl files to the new GNUWin environment?* [AR# 41136] – 2011/2012

[34] Xiph.org – *Xiph.org Video Test Media (derf's collection)* – http://media.xiph.org/video/derf/

[35] **Javier Mora**, Ángel Gallego, Andrés Otero, Eduardo de la Torre, Teresa Riesgo – *Noise-agnostic Adaptive Image Filtering without Training References on an Evolvable Hardware Platform* – 2013 Conference on Design and Architectures for Signal and Image Processing (DASIP) – Cagliari (Italy), 2013

[36] Rubén Salvador, Andrés Otero, **Javier Mora**, Eduardo de la Torre, Teresa Riesgo, Lukáš Sekanina – *Implementation techniques for evolvable HW systems: virtual VS. dynamic reconfiguration* – 22nd International Conference on Field Programmable Logic and Applications (FPL), pp.547–550 – Oslo (Norway), 2012

[37] Ángel Gallego, **Javier Mora**, Andrés Otero, Rubén Salvador, Eduardo de la Torre, Teresa Riesgo – *A Novel FPGA-based Evolvable Hardware System based on Multiple Processing Arrays* – Reconfigurable Architectures Workshop (RAW) at 2013 IEEE 27th International Symposium on Parallel & Distributed Processing Workshops and PhD Forum (IPDPS) – Boston (MA, USA), 2013

[38] Ángel Gallego, **Javier Mora**, Andrés Otero, Eduardo de la Torre, Teresa Riesgo – *A Scalable Evolvable Hardware Processing Array* – 2013 International Conference on ReConFigurable Computing and FPGAs (ReConFig) – Cancun (Mexico), 2013 *(acceptance pending)*

[39] Andrés Otero, Eduardo de la Torre, Teresa Riesgo – *Dreams: A Tool for the design of Dynamically Reconfigurable Embedded and Modular Systems* – 2012 International Conference on Reconfigurable Computing and FPGAs (ReConFig), pp.1–8 – Cancun (Mexico), 2012

[40] Simen Gimle Hansen, Dirk Koch, Jim Tørresen – *High Speed Partial Run-Time Reconfiguration Using Enhanced ICAP Hard Macro* – 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), pp.174–180 – Shanghai (China), 2011